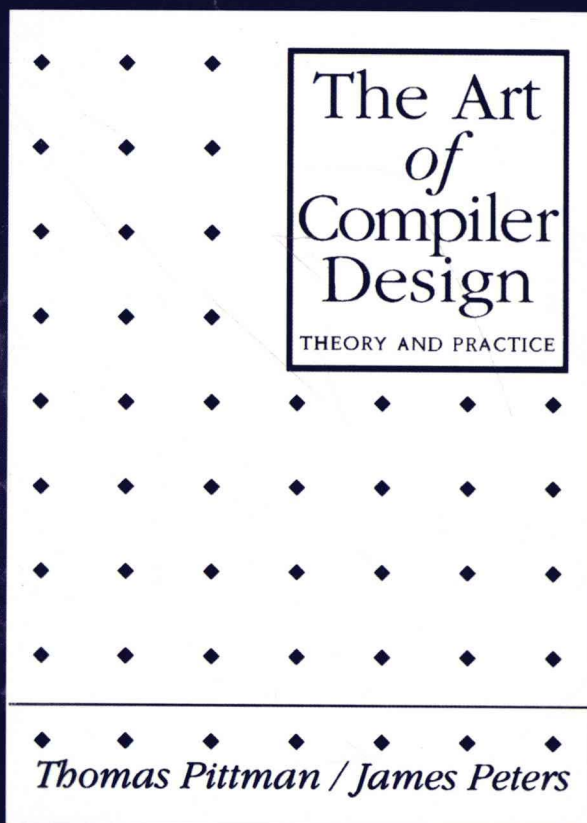


编译程序设计艺术 理论与实践

(美) Thomas Pittman James Peters 著 李文军 高晓燕 译



The Art of Compiler Design
Theory and Practice

编译程序设计艺术 理论与实践

本书详细介绍了编译程序设计中的词法分析（扫描程序）、语法分析（分析程序）、语义分析（约束程序）、中间代码优化以及代码生成等内容。作为颇受好评的编译原理优秀入门教材，本书的最大特色是在全书贯穿了一种基于文法的指导思路：在语法分析阶段，该书遵循了一般教材采用的上下文无关文法；在语义分析阶段，采用以上下文无关文法为基础的属性文法；而在代码优化和代码生成阶段，则采用了变换属性文法。书中最后还给出变换属性文法的一种自编译实现。此外，本书还探讨了面向不同计算机体系结构的代码生成技术以及非过程式语言的编译问题。

本书适合作为高等院校计算机科学与技术、软件工程以及相关专业编译原理课程的教学参考书；同时也可供计算机语言及其处理技术爱好者参考。

本书特点

- 坚定不移地扎根于文法，一开始就介绍文法和语言识别器之间的理论关系，然后贯穿全书将文法技术应用到编译程序设计的每一方面。
- 统一将实用的属性文法作为编译程序语义的载体，坚持这一立场自然会产生一个完全由属性文法定义的、可编译其自身的“编译程序—编译程序”。
- 具有非常实用的特征，编译程序的“设计”必须以属性文法定义，而编译程序的“构造”则需要可执行的代码，并且每一个重要的理论原则均需通过一种真实程序设计语言的大量代码清单加以阐明，不断展示文法与机器代码之间极其自然的关系。
- 选择Modula-2作为演示代码的程序设计语言，旨在概念抽象与具体效率之间取得平衡。



客服热线：(010) 88378991, 88361066
购书热线：(010) 68326294, 88379649, 68995259
投稿热线：(010) 88379604
读者信箱：hzjsj@hzbook.com

PEARSON

www.pearsonhighered.com

华章网站 <http://www.hzbook.com>

网上购书：www.china-pub.com

封面设计：金锡 林

上架指导：计算机 编译原理

ISBN 978-7-111-28810-7



定价：55.00元

TP314
P649

机 科 学

编译程序设计艺术 理论与实践

(美) Thomas Pittman James Peters 著 李文军 高晓燕 译

TP314
P649

The Art of Compiler Design
Theory and Practice



机械工业出版社
China Machine Press

本书详细介绍了编译程序设计中的词法分析（扫描程序）、语法分析（分析程序）、语义分析（约束程序）、中间代码优化以及代码生成等内容。作为颇受好评的编译原理经典教材，本书的最大特色是在全书贯穿了一种基于文法的指导思路，语义分析、代码优化、代码生成等均采用文法定义其规格说明，并且最后给出了变换属性文法（TAG）的一种自编译实现。此外，本书还探讨了面向不同计算机体系结构的代码生成技术以及非过程式语言的编译问题。

本书适合作为高等院校计算机科学与技术、软件工程以及相关专业编译原理课程的教学参考书，同时也可供计算机语言及其处理技术爱好者参考。

Simplified Chinese edition copyright © 2010 by Pearson Education Asia Limited and China Machine Press.

Original English language title: *The Art of Compiler Design: Theory and Practice* (ISBN 0-13-048190-4) by Thomas Pittman and James Peters, Copyright © 1992.

All rights reserved.

Published by arrangement with the original publisher, Pearson Education, Inc., publishing as Prentice Hall.

本书封底贴有 Pearson Education（培生教育出版集团）防伪标签，无标签者不得销售。

版权所有，侵权必究

本书法律顾问 北京市展达律师事务所

本书版权登记号：图字：01-2009-1591

图书在版编目（CIP）数据

编译程序设计艺术：理论与实践 /（美）皮特曼（Pittman, T.），皮特斯（Peters, J.）著；李文军，高晓燕译. —北京：机械工业出版社，2010.1

（计算机科学丛书）

书名原文：The Art of Compiler Design: Theory and Practice

ISBN 978-7-111-28810-7

I. 编… II. ①皮… ②皮… ③李… ④高… III. 编译程序—程序设计 IV. TP314

中国版本图书馆 CIP 数据核字（2009）第 206766 号

机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码 100037）

责任编辑：迟振春

三河市明辉印装有限公司印刷

2010 年 1 月第 1 版第 1 次印刷

184mm×260mm·22 印张

标准书号：ISBN 978-7-111-28810-7

定价：55.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：（010）88378991；88361066

购书热线：（010）68326294；88379649；68995259

投稿热线：（010）88379604

读者信箱：hzsj@hzbook.com

出版者的话

文艺复兴以降，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域取得了垄断性的优势；也正是这样的传统，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅筹划了研究的范畴，还揭示了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短的现状下，美国等发达国家在其计算机科学发展的几十年间积淀和发展的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起到积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章分社较早意识到“出版要为教育服务”。自1998年开始，华章分社就将工作重点放在了遴选、移译国外优秀教材上。经过多年的不懈努力，我们与Pearson, McGraw-Hill, Elsevier, MIT, John Wiley & Sons, Cengage等世界著名出版公司建立了良好的合作关系，从他们现有的数百种教材中甄选出Andrew S. Tanenbaum, Bjarne Stroustrup, Brian W. Kernighan, Dennis Ritchie, Jim Gray, Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman, Abraham Silberschatz, William Stallings, Donald E. Knuth, John L. Hennessy, Larry L. Peterson等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及珍藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力襄助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专程为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近两百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍。其影印版“经典原版书库”作为姊妹篇也被越来越多实施双语教学的学校所采用。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图书有了质量的保证。随着计算机科学与技术专业学科建设的不断完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都将步入一个新的阶段，我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的重要帮助。华章分社欢迎老师和读者对我们的工作提出建议或给予指正，我们的联系方式如下：

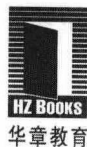
华章网站：www.hzbook.com

电子邮件：hzjsj@hzbook.com

联系电话：(010) 88379604

联系地址：北京市西城区百万庄南街1号

邮政编码：100037



译者序

长期以来,编译原理一直是计算机相关专业本科生的专业主干课程。提起编译原理的英文原版教材,许多人自然想到龙书(A. Aho 等人所著《*Compilers: Principles, Techniques, and Tools*》,俗称 Dragon Book)、虎书(A. Appel 所著《*Modern Compiler Implementation in C/Java/ML*》,俗称 Tiger Book)、鲸书(S. Muchnick 所著《*Advanced Compiler Design and Implementation*》,俗称 Whale Book)等经典著作,其中鲸书侧重于编译程序的后端,并不适合第一门编译原理课程使用。此外,较有影响力的教材还有 K. Louden 所著《*Compiler Construction: Principles and Practice*》、C. Fischer 等人所著《*Crafting a Compiler with C*》、K. Cooper 等人所著《*Engineering a Compiler*》以及 A. Holub 所著《*Compiler Design in C*》等。

与上述流行教材相比,由美国阿肯色大学 T. Pittman 和 J. Peters 合著的本书的最大特色,是在全书贯穿了一种基于文法的指导思路:在语法分析阶段,该书遵循了一般教材采用的上下文无关文法;在语义分析阶段,采用以上下文无关文法为基础的属性文法;而在代码优化和代码生成阶段,则采用了变换属性文法。书中最后还给出变换属性文法的一种自编译实现。由于将文法技术贯穿于编译程序设计的每一方面,设计人员可以在规格说明层次定义一种语言的语法和语义,这将有益于编译程序的实现(无论采用手工方式,还是基于生成工具的自动方式)。例如,书中“Tiny BASIC 解释程序”和“Micro-Modula 美化打印工具”这两个例子很好地展示了这一方法的强大能力,以及独立于具体实现语言的特性。

本书虽然扎根于文法,但又不至于赘述文法。例如,与很多教材不同的是,本书基于下推自动机模型讲解 LL(k)分析技术,有助于读者更好地理解编译程序设计与其理论基础之间的关联。又如,定义正则语言的形式化工具包括正则表达式、正则文法(左线性和右线性)、有穷状态自动机(确定的和不确定的)等多种等价方式,它们之间的等价转换是编译原理的重要理论基础;本书在词法分析部分重点介绍了从定义词法规则的正则表达式(或正则文法)到约简的确定有穷状态自动机这一变换路径上的各个环节,而不像一些教材那样将形式语言和自动机理论的许多内容未加以精心筛选就照搬到编译原理课程中。

尽管本书在编译程序设计的应用技术与理论基础之间的折中可圈可点,但也有未尽人意之处,例如第 7 章关于 LR(k)分析技术的介绍可能会令许多初学者“知其然而不知其所以然”。在介绍编译程序后端的许多内容时,本书也因为过于简略并且缺少具体例子的解说,给读者理解相关技术造成困难。

鉴于本书编写年代较早,书中未涉及近十多年来编译程序的原理、方法、技术、工具等方面的最新进展,譬如面向对象程序设计语言的编译、垃圾收集机制的实现、许多代码优化技术等,这是读者使用本教材时应注意的地方。但这并不妨碍本书作为第一门编译原理课程的优秀入门教材或参考书,这些不足可通过其他较新的教材弥补。

中山大学计算机科学系软件工程实验室(selab@sysu)的周晓聪、乔海燕、罗达、李曦、朱建伟、梁焯佳、吴梓彦仔细阅读了译稿,并提出了宝贵的意见和建议,译者对此表示衷心感谢。由于译者水平所限,书中谬误之处在所难免,恳请广大读者不吝批评指正。

李文军 高晓燕

2009 年 9 月于广州康乐园

前言


实用且复杂的现代信息系统并不是从累积的偶然事件中发展起来的。编译程序作为一个实用的信息系统，只有经过精心设计，才能以简单的措辞理解其复杂性。不仅未经设计的编译程序会有功能缺陷，编译程序理论本身还依赖于它与两种理论之间的一种微妙关系，其中一种理论是源自人类自然语言的语言学原理，另一种理论是将计算机看作有穷状态自动机。理解并利用这一关系需在文法理论的基本原理方面有扎实的根基，并熟练掌握其机械化实现过程。因而编译程序设计的教学亦涉及一个复杂的信息系统，需精心设计才能以连贯、合理的次序展示相关概念，从而让学生体会到编译程序设计的内容既是相关的，也是可管理的。

本书并非一本试图以所有可能的方法去构造所有可能的编译程序的百科全书，而是依次介绍编译程序设计中的最基本问题；其内容的深度足以让勤奋的学生有能力通过手工方式或使用编译程序生成工具（亦或两种方式之结合），构造实用且高效的编译程序。更重要的是，学生将明白这些工具是如何工作的，以及为什么必须以某种方式书写文法方可取得预期的结果。这正是设计的原则。因此，尽管大多数现代分析程序生成工具采用自底向上分析技术，但本书深入研究有更多限制的自顶向下分析理论，然后才利用相对简短（但完整）的一章内容学习自底向上分析程序。本书每一章的目标都是先灌输对基本概念的理解，然后再将这些概念应用到实践中。

与同类教材相比，本书在四个重要方面有显著特色。第一个特色是，它坚定不移地扎根于文法，一开始就介绍文法和语言识别器之间的理论关系，然后贯穿全书将文法技术应用到编译程序设计的每一方面。第二个特色是，统一将实用的属性文法作为编译程序语义的载体，坚持这一立场自然会产生一个完全由属性文法定义的、可编译其自身的“编译程序—编译程序”，这正是本书最后一章的重点。第三个特色是，具有非常实用的特征，编译程序的设计必须以属性文法定义，而编译程序的构造则需要可执行的代码，并且每一个重要的理论原则均需通过一种真实程序设计语言的大量代码清单加以阐明，不断展示文法与机器代码之间极其自然的关系。第四个特色是，选择 Modula-2 作为演示代码的程序设计语言，旨在概念抽象与具体效率之间取得平衡；与 C 语言等更低级的语言相比，应用本书后几章所介绍的优化技术能以更低的开销更显著地提高 Modula-2 程序的效率。


本书可用于一学期的编译原理入门课程，重点介绍前 6 章或前 7 章。本书亦可用于整学年的学习，更好地涉猎更多的高级课题。一学期的课程学习可安排在半年或一个季度的时间内完成；经过不断优化和实际教学检验，本书循序渐进地布置学生实验项目，并在期末完成一个可工作的 Itty Bitty Modula 编译程序，该编译程序可在最终的实验项目中用于编译另一个分析程序，例如第 6 章最后概述的美化打印工具或 Tiny BASIC 语言解释程序。

书中的某些章节和练习被设计为任选的。虽然我们坚信编译程序设计需要扎实的理论基础，但根据时间安排或学生能力等，某些章节中的一些趣味数学内容可略过，这些内容在其

页边空白处用一个“教师授课”图标  注明。另一些章节虽与编译程序设计的主要问题

VI

密切相关，但由于本书的定位是初学编译原理的学生，他们的水平难以理解这些内容，因而

在这些章节的页边空白处标注了一个“石中剑”图标 。类似地，有些问题富有挑战性，读者解决这些问题有助于更好地理解编译程序设计的错综复杂，但不在这些问题上花费额外的时间和精力，也不影响对课程内容的良好理解，因而在这些问题的页边空白处也标注了“石中剑”图标。

尽管无法逐一向所有对本教材作出贡献的人致谢，我们仍要首先感谢 Brad Blaker，没有他的鼓励和早期协助，本书将不会面世；感谢 Bill Hankley、Austin Melton 和堪萨斯州立大学耐心的同学们，他们坚持使用了早期的书稿；感谢 Frank DeRemer 孕育了书中的许多思想。Thom Boyer、Dick Karpinski、Brian Kernighan、Marvin Zelkowitz、Wayne Citrin、Norman C. Hutchinson、Johnson M. Hart、Bernhard Weinberg、Will Gillett，特别是 Dean Pittman、Chota 和 Dave Schmidt 等人提出的意见和建议对本书的最终定稿有莫大帮助。

Thomas Pittman

James Peters

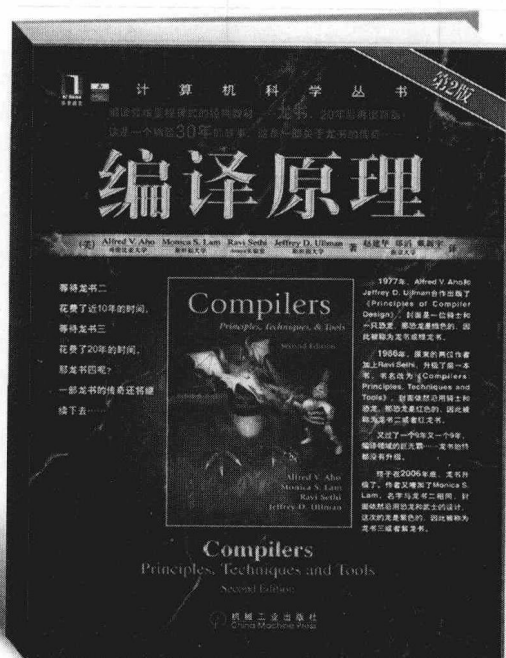
编译领域里程碑式的经典著作——**龙书**,

20年后终于出版新版!

这是一个延绵30年的故事,

这是一部关于龙书的传奇!

最新版本,增添两章节内容,使龙书地位更权威!



【原书名】 Compilers: Principles, Techniques, and Tools (2nd Edition)
【原出版社】 Addison Wesley
【书号】 978-7-111-25121-7
【作者】 (美) Alfred V. Aho 等
【译者】 赵建华 等

编译领域里程碑式的经典著作——龙书, 20年后终于出版新版! 这是一个延绵30年的故事, 这是一部关于龙书的传奇! 最新版本, 增添两章节内容, 使龙书地位更权威!

本书是编译领域无可替代的经典著作, 被广大计算机专业人士誉为“龙书”。本书上一版自1986年出版以来, 被世界各地的著名高等院校和研究机构(包括美国哥伦比亚大学、斯坦福大学、哈佛大学、普林斯顿大学、贝尔实验室)作为本科生和研究生的编译原理课程的教材。该书对我国高等计算机教育领域也产生了重大影响。

第2版对每一章都进行了全面的修订, 以反映自上一版出版20多年来软件工程。程序设计语言和计算机体系结构方面的发展对编译技术的影响。本书全面介绍了编译器的设计, 并强调编译技术在软件设计和开发中的广泛应用。每章中都包含大量的习题和丰富的参考文献。

目 录

出版者的话

译者序

前言

第1章 编译程序理论概述.....1

- 1.1 简介.....1
- 1.2 语言与翻译程序.....1
- 1.3 文法的作用.....2
- 1.4 若干例子.....4
- 1.5 编译程序的结构.....6
 - 1.5.1 词法分析.....7
 - 1.5.2 字符串表.....8
 - 1.5.3 语法分析.....9
 - 1.5.4 约束.....9
 - 1.5.5 符号表.....9
 - 1.5.6 代码生成.....9
 - 1.5.7 优化.....10

符号.....11

缩略词.....11

关键术语.....11

练习.....12

复习小测验.....12

编译程序实验项目.....13

进一步阅读.....13

第2章 文法：乔姆斯基层次.....14

- 2.1 简介.....14
- 2.2 文法.....14
 - 2.2.1 字母表与串.....14
 - 2.2.2 非终结符与产生式.....15
 - 2.2.3 若干文法例子.....15
- 2.3 乔姆斯基层次.....18
- 2.4 文法及其机器.....19
 - 2.4.1 图灵机.....19
 - 2.4.2 线性有界自动机.....20
 - 2.4.3 下推自动机.....20
 - 2.4.4 删除空产生式.....21

2.4.5 比较上下文无关文法和上下文

敏感文法.....22

2.4.6 有穷状态自动机.....22

2.5 空串与空语言.....23

2.6 规范推导.....23

2.7 二义性.....24

2.8 文法思维的艺术.....25

2.8.1 有穷状态自动机的局限性.....26

2.8.2 上下文无关文法的计数.....27

2.8.3 对上下文敏感.....29

小结.....30

符号.....31

缩略词.....31

关键术语.....31

练习.....32

复习小测验.....34

编译程序实验项目.....35

进一步阅读.....35

第3章 扫描程序和正则语言.....37

3.1 词法分析简介.....37

3.2 正则表达式.....37

3.2.1 正则表达式代数.....39

3.2.2 正则表达式的形式化特性.....40

3.3 文法与正则表达式的转换.....41

3.4 有穷状态自动机.....44

3.5 不确定的有穷状态自动机.....45

3.6 将文法转换为自动机.....46

3.7 自动机的转换.....48

3.7.1 删除空环路.....49

3.7.2 删除空变迁.....50

3.7.3 自动机的确定化.....51

3.7.4 自动机的约简.....52

VIII

3.8 将自动机转换为文法	53	4.7.1 使用 TAG 编译程序	90
3.9 左线性文法	54	4.7.2 使用 YACC	92
3.10 在计算机上实现有穷状态自动机	54	4.8 递归下降分析程序	92
3.11 扫描程序的特殊实现问题	59	4.9 递归下降分析程序作为下推自动机	95
3.11.1 输入字母表的大小	59	小结	95
3.11.2 扫描程序自动机中的停机状态	60	缩略词	96
3.11.3 过滤空格与注释	60	关键术语	96
3.11.4 单词的输出	61	练习	97
3.12 字符串表的实现	62	复习小测验	101
3.12.1 基于线性查找的实现	63	编译程序实验项目	101
3.12.2 基于散列表的实现	64	进一步阅读	102
3.12.3 基于查找树的实现	66	第 5 章 语义分析与属性文法	103
3.12.4 不同实现的性能比较	69	5.1 简介	103
3.13 保留字	69	5.2 属性文法	103
3.14 使用扫描程序生成工具	70	5.2.1 继承属性和综合属性	104
小结	70	5.2.2 属性值流	107
符号	71	5.3 非终结符作为属性求值函数	108
缩略词	71	5.4 符号表作为属性	109
关键术语	71	5.5 Micro-Modula 的属性文法	110
练习	72	5.6 在 TAG 编译程序中使用属性	113
复习小测验	74	5.7 作用域与标识符类别	114
编译程序实验项目	74	5.7.1 标识符作用域的文法	114
进一步阅读	75	5.7.2 标识符作用域例子分析	116
第 4 章 分析程序和上下文无关语言	76	5.7.3 符号表的其他问题	120
4.1 简介	76	5.8 在递归下降中实现属性	121
4.2 下推自动机	76	5.9 实现符号表	122
4.2.1 停机条件的等价性	78	小结	123
4.2.2 根据上下文无关文法构造下推自动机	79	符号	123
4.3 LL(k)条件	81	关键术语	123
4.3.1 First 和 Follow 集	82	练习	124
4.3.2 选择集	83	复习小测验	126
4.4 左递归	85	编译程序实验项目	126
4.5 公共左因子	86	进一步阅读	126
4.6 为上下文无关文法扩展正则表达式运算符	88	第 6 章 语法制导代码生成	128
4.7 使用分析程序生成工具	89	6.1 简介	128
		6.2 计算机硬件体系结构	128
		6.3 栈机器的表达式求值	130
		6.4 Itty Bitty 栈机器	131

6.5 带属性的代码生成	134	7.7 LALR(k)分析程序	174
6.5.1 运算符优先级与结合性质	137	7.8 自底向上分析程序的实现	175
6.5.2 程序结构的语义	138	7.9 出错恢复	177
6.5.3 向前分支问题	139	7.10 LR 分析程序中的属性求值	177
6.6 过程和函数的代码生成	143	小结	178
6.7 块结构的栈帧管理	144	关键术语	179
6.7.1 帧与帧指针	144	练习	180
6.7.2 静态链与动态链	145	复习小测验	180
6.7.3 帧指针的 Display 向量	146	编译程序实验项目	181
6.8 其他数据类型	148	进一步阅读	181
6.9 结构化数据类型	149	第 8 章 变换属性文法	182
6.9.1 指针类型	150	8.1 简介	182
6.9.2 记录结构	151	8.2 程序的树表示	182
6.9.3 数组的语义	151	8.3 树变换文法	183
6.10 其他数据结构	153	8.3.1 非生成的文法	186
6.11 Itty Bitty 栈机器的输入和输出	154	8.3.2 一个 TAG 例子	187
6.12 语法制导语义的局限	154	8.3.3 求值次序	188
6.13 手工编写编译程序的代码生成	155	8.3.4 信息流与存储	188
6.14 语法制导语义的应用	155	8.3.5 带树值的属性	189
6.14.1 Tiny BASIC 解释程序	155	8.3.6 不确定的分析	191
6.14.2 Micro-Modula 美化打印工具	156	8.4 组合串文法与树文法	191
小结	158	8.5 TAG 中的类型检查	192
关键术语	158	8.6 基于变换的代码优化	193
练习	159	8.6.1 数据流分析	193
复习小测验	160	8.6.2 数据流分析中使用属性文法	197
编译程序实验项目	160	8.7 中间代码树表示的替代方案	198
进一步阅读	160	8.7.1 四元式的数据流	199
第 7 章 自底向上分析程序的自动化		8.7.2 循环的数据流分析	200
设计	162	8.8 实用优化变换综述	203
7.1 简介	162	8.8.1 模拟执行优化的类别	204
7.2 LR(k)分析程序	165	8.8.2 常量折叠分析	204
7.2.1 构造 LR(k)状态机	166	8.8.3 使用值编号检测公共子表达式	208
7.2.2 一个 LR(2)分析程序	168	8.8.4 左移动提升	211
7.2.3 归约与移进操作	168	8.8.5 右移动提升	212
7.3 冲突	169	8.8.6 无用代码以及其他从右到左的数据流分析	216
7.4 例子: 文法 G_2 的冲突解析	169	8.8.7 数学等式与代码选择	216
7.5 在栈中保存状态	171	8.8.8 循环结构分析	217
7.6 其他 LR(k)分析程序: SLR	172		

8.9 实现抽象语法树	220	9.6.2 窥孔优化	277
8.10 实现 TAG 驱动树的变换	228	小结	277
小结	231	缩略词	277
符号	232	关键术语	278
缩略词	232	练习	278
关键术语	232	复习小测验	279
练习	233	编译程序实验项目	279
复习小测验	234	进一步阅读	279
编译程序实验项目	234	第 10 章 非过程式语言	281
进一步阅读	234	10.1 简介	281
第 9 章 代码生成与优化	237	10.2 应用式语言的编译	282
9.1 简介	237	10.2.1 Lisp 语言的一些概念	283
9.2 循环优化	237	10.2.2 尾递归	284
9.2.1 循环的范围分析	238	10.2.3 实现一个应用式语言的编译 程序	285
9.2.2 归纳变量	239	10.3 变换属性文法的编译程序	293
9.2.3 循环展开	240	10.3.1 TAG 编译程序的组成部分	293
9.3 寄存器与内存分配	241	10.3.2 文法中的迭代运算符	294
9.3.1 寄存器分配算法	242	10.3.3 向用户报告语法错误	295
9.3.2 表达式中的寄存器分配	243	10.3.4 自动构造扫描程序	296
9.3.3 更好的寄存器分配数据流 分析	257	10.3.5 TAG 编译程序的语法分析	300
9.3.4 循环的寄存器分配	257	10.3.6 树变换	305
9.3.5 寻址模式	258	10.3.7 语法错误停机	307
9.3.6 分支寻址选择	259	小结	307
9.3.7 分支链	264	关键术语	308
9.4 代码生成的复杂性	265	练习	308
9.4.1 指令选择	266	复习小测验	309
9.4.2 强度削弱	268	进一步阅读	309
9.5 专用指令	269	附录 A Itty Bitty Modula 语法图	311
9.5.1 RISC 和流水线处理器调度	269	附录 B TAG 编译程序的 TAG	313
9.5.2 向量处理器	272	附录 C Itty Bitty 栈机器的指令集	335
9.6 代码优化的变形	276	附录 D 四种计算机的代码生成表	339
9.6.1 代码优化的分类	276		

第1章 编译程序理论概述

本章旨在：

- 综述编译的目的和方法
- 介绍语言规格说明中的文法概念
- 纵览一个编译程序的结构
- 介绍编译程序使用的基本数据结构
- 区别词法分析和语法分析
- 综述编译程序的前端和后端

1.1 简介

在计算机科学中，编译程序设计是为数不多的抽象理论彻底改变程序编写方式的领域之一。最早的编译程序大多采用基于直觉与经验的非正规方法编写，使用传统的程序设计技术。文法驱动的分析程序的出现改变了这一切。我们现在所看到的真正编译程序，全是先用上下文无关文法书写，然后再机械地转换为代码。

本书讨论现代编译程序的设计，因而势必涉及文法。一个优秀编译程序的每一部分均以某种方式与定义它的文法联系在一起。本书将展示一个编译程序的文法规格说明就是该编译程序，只不过采用了非常高级的语言编写；本书还将说明如何利用文法编写一个编译程序，以及如何编写文法的编译程序，将文法编译为编译程序。设计是由文法理论驱动的，因而设计是清晰的且易于实现的。聪明的读者可在数日内从本书学会如何为一门简单但实际可行的程序设计语言编写一个完整的编译程序。

1.2 语言与翻译程序

类似于英语、法语、俄语等自然语言，计算机语言为信息交流定义了一种将单词组织为句子的方式。自然语言可用于交流内心的感觉、外界的事实、关于这些事实和感觉的疑问、听众或读者必须遵从的指示等，而计算机语言通常仅限于表达接收这些语言的机器必须遵从的命令。

自然语言限制了可表达的形式，但并未限制可表达的内容。例如，在英语中可以说“Peter hit the ball”，但不能说“ball Peter the hit”，前者在文法上是正确的，而后者不正确；类似地，在法语中文法正确的说法是“Pierre frappa la balle”。懂得上述两种语言的读者可能立即意识到上述英语句子和法语句子说的是同一件事，即它们具有相同的含义，但对于仅熟悉其中一种语言的读者就未必是显而易见的。单词“frappa”对于说英语的人是没有意义的，单词“hit”在法语中也没有任何含义。即使通过字典查到这两个动词的含义，各门语言的文法还定义了动词的时态，这些时态由单词的形式指明：上述两个动词都是过去时。

当说英语的人想与说法语的人交流，并且两人均不了解对方语言时，有必要带上一位翻译人员。在自然界中，翻译人员收到用一种语言表达的信息后，用另一语言重复这一信息。将英语翻译为法语的翻译人员读到“Peter hit the ball”时，将写出“Pierre frappa la balle”。如果翻译人员遇到“ball Peter the hit”这种表示时，他可能回复说这一句子没有意义；因为它在英语

中没有意义，所以无法将此句翻译成一个有意义的法语句子。

编译程序是一个扮演着翻译人员角色的计算机程序，它读入某一计算机语言的语句，如果这些语句在该语言中是有意义的，则将它们翻译为含义相同的另一计算机语言的语句。有一些规则定义了每一种语言中什么是有意义的，编译程序运用这些规则确定其输入是否有意义，并保证其输出是有意义的。使用计算机语言编写的一系列语句组成一个程序；编译程序将一种计算机语言（称为源语言）的程序翻译为另一种计算机语言（称为目标语言）的程序（即语句序列）。

实际上存在多种计算机语言以及计算机语言的翻译程序。最简单的翻译程序是读入一种用简单计算机语言书写的单词，然后将这些单词直接翻译为计算机指令代码中的数字。这种翻译程序称为汇编程序，其源语言称为汇编语言。这一命名缘于大多数机器指令由几个部分组成，汇编语言采用不同的单词或数字表示每一部分，汇编程序则将这些部分组装成一个数值代码。汇编程序充其量不过是一个表查找例程，在表中查找源语言中每一单词对应的数字表示，并将查找结果输出，作为目标语言程序的组成部分。汇编语言通常使得程序员可准确、直接地访问计算机硬件的每一功能；然而与大多数其他计算机语言相比，使用汇编语言编写正确的程序是相当困难的。

“编译程序”这一术语一般留给更复杂的语言使用，其中源语言单词和目标语言之间不存在简洁而直接的对应关系。大多数编译程序的目标语言通常也是机器语言，这与汇编程序的目标语言相同；计算机语言翻译程序的目的固然是简化创建机器语言程序的过程，但许多早期的编译程序、甚至一些现代的编译程序都先编译为汇编语言，然后借助于汇编程序完成汇编语言到机器语言的翻译。然而，编译程序的源程序通常是所谓的高级语言（简称 HLL），高级语言的特点是更接近于问题求解的表示法，而不是机器语言。例如，对于商业应用而言，COBOL（COmmon Business Oriented Language）语言采用会计人员和中层管理人员易于理解的术语；而科学计算问题往往表述为公式，FORTRAN（FORmula TRANslator）语言被认为更适合表达这些公式。现在一些程序员更喜欢一种语言既有高级语言中更抽象的结构，又带有汇编程序支持的低层控制，为此他们使用 C 语言（得名于它作为早期 B 语言的下一代语言）。程序设计方法学的最新进展提倡模块化软件设计，Modula-2 语言非常强调这一特性。

解释程序在某些方面类似于翻译程序，它也读入一个高级语言的程序，但立即进行翻译，就好像翻译人员的口译一样即可听到和理解。编译程序将一个计算机程序翻译为稍后再执行的机器代码，而解释程序则边读入、边执行程序。从某种意义上讲，解释程序从来没有真正地完成翻译过程，就好像前述例子中的翻译人员，当他听到“*Peter, hit the ball !*”的指示后，并不回应“*Pierre, frappa la balle !*”，而是径自走过去击球。由于解释程序不必关注目标语言，它可以比编译程序更快地处理一行输入程序。解释程序必须反复读入其输入程序以计算结果，但编译程序仅将输入程序翻译一次。编译程序首次运行计算机程序并得到结果要花费更长时间，但后续的运行则远快于解释程序，因为此时不再需要额外的翻译时间。

本书的重点放在编译程序的设计上，但某些练习也包含了解释程序。

1.3 文法的作用

我们从英语或法语这类自然语言中学到的特性之一是这些语言的文法。一门语言的文法定

义了该语言中句子的正确形式。例如，英语中可能有一些如下的规则：

sentence	→	noun-phrase verb noun-phrase
verb	→	"hit"
noun-phrase	→	article noun
	→	proper-name
article	→	"a" "the"
noun	→	"ball" "bat"
proper-name	→	"Peter"

上述文法表明，一个句子可由两个名词短语中间加上动词组成，即文法中由单词 sentence 表达的抽象概念“句子”可重写或替换为三个抽象概念“名词短语”、“动词”和另一“名词短语”组成的序列。一个名词短语可以是 Peter 这一类专有名词，也可以是带有冠词 the 或 a 的普通名词 ball。本例中的动词自然是 hit。

类似地，计算机语言的文法通过指定语言中的抽象概念如何重写为更具体的符号序列，定义了该语言中句子的正确形式。文法中的每一重写规则均表示为一个单词通过一个箭头连接到一个或多个单词。当然，这里所说的“句子”会在计算机语言中给出细致的定义。

从单词 sentence 开始，上述小文法不仅可产生句子“*Peter hit the ball*”，还可产生“*a ball hit Peter*”和有些荒谬的“*Peter hit Peter*”等句子。当文法定义了多个选项时（要么用多个箭头，要么用“或”分隔符“|”），可选择其中任一选项用于重写箭头左边名字的一次出现。一个文法产生的语言，是由逐一选择每一可能选项的所有组合所产生的全部句子的集合；上述文法的语言恰好有 25 个可能的句子。

一门程序设计语言通常由两个不同的文法定义，其中一个文法定义了语言的单词，另一文法定义了这些单词是如何组合在一起的。类似地，文法还可用于书写目标语言的定义；最新的研究已开始关注如何从源语言和目标语言的文法自动构造一个编译程序，但研究成果良莠不齐。因而，本书仍遵循更传统的编译程序设计方法学，使用属性文法明确地定义翻译工作是如何进行的。定义一个编译程序时可使用多种文法，每一文法均定义了编译程序中某一部件的功能。

最重要的一类文法是短语结构文法，它定义了一个编译程序或解释程序的核心部件，该部件称为分析程序。短语结构文法定义了计算机语言中的“单词”如何组织在一起，形成一个合乎语法的程序。“分析”是自然语言研究中的一个术语，描述了根据文法的形式分析该语言中一个句子的过程；考虑计算机语言时，我们以完全相同的方式使用该术语。上述关于英语语言的小文法是一个短语结构文法。

第二重要的文法通常用于定义计算机语言中单词的正确形式或拼写方法，该文法称为词汇文法 (lexical grammar)，得名于“单词”一词的拉丁文。编译程序中，分析输入程序中各个单词的部分称为扫描程序。我们可按如下方式为英语单词定义一个词汇文法：

word	→	letter word
	→	letter

其中，每个 letter 表示英文字母表中 26 个字母之一。尽管该文法会生成一大堆毫无意义的单词，但它说明了生成一个非常大的语言未必需要一个复杂的文法。实际上该文法定义的语言是无穷

的：只要我们选择第一个选项，在一个单词之前再添加一个字母，则可得到任意长度的单词。

在词汇文法和短语结构文法上附加一些值（称之为属性）以及属性求值函数、断言等，即可定义一个编译程序的翻译过程；还可使用其他属性文法定义编译程序如何改进被编译程序的时间或空间性能。我们可利用断言避免诸如“*Peter hit Peter*”这类没有意义的句子，例如由断言约束文法产生的每一句子中仅含最多一个专有名词。

本书重点讨论基于文法的编译程序构造方法。我们认为文法是一个编译程序的高级语言规格说明，实际上它也是一个编译程序。换言之，所有编译程序设计都应深入探讨如何编写一个文法，以定义语言的各组成部分及其翻译过程；只要妥善完成了这一任务，编译程序设计的其余部分就都是机械的且可自动化的。本书还详细介绍了根据文法自动构造一个编译程序的工具，由于这些工具本身也是编译程序，因此它们的设计也成为本书的重要部分。

1.4 若干例子

我们关注于将编译程序的构造作为一个语言规格说明的逻辑扩展，这种做法的好处之一是让我们更容易体会到由文法定义的强类型语言的清晰设计。FORTRAN 语言的设计采用了较为随意的方式表示数学公式和简单控制结构。由于没有用文法的规格说明驱动语言的设计，为 FORTRAN 编译程序编写一个确定的且快速的扫描程序是极其痛苦的。例如，考虑以下两行语句，它们在 FORTRAN 语言的最初版本中都是合法的（注意，空格在 FORTRAN 语言中是无意义的，故可全部省略）：

```
DO10K=1.9
DO10K=1,9
```

第一行是一条赋值语句，将实数值 1.9 赋值给浮点类型变量 DO10K；第二行是一个循环结构的开头部分，该循环以一条标号为 10 的语句结尾，其控制变量 K 从 1 步进到 9。这里未能识别一行带语句标号 10 的语句作为循环的终结符，因而编译程序无法判断第一行语句是否本意写成第二行的样子而在键盘录入时输错了（反之亦然）。实际上，已有报导说首架金星探测器陨落坠地的损失正是归咎于这样的一个程序设计错误（逗号与英文句号如此误用完全改变了控制太空探测器的 FORTRAN 程序的含义）。此外，上述例子中编译程序必须检查完该语句中除最后一个字符之外的所有字符，才能开始考虑如何处理第一行语句，即该语句是以关键字 DO 开头，还是以实数变量的标识符开头。

FORTRAN 语言是每一位语言设计人员的首选攻击对象。我们仅论及该语言的另一经典难题，这一例子还说明了这些困难原本可由合理且无二义的文法规格说明消除。在以下 4 行无意义的程序中，第 10 行是一条赋值语句，还是一条输出语句的格式说明？

```
DIMENSION FORMAT(72)
10  FORMAT(X3H)=(I5)
WRITE(6,10)K
END
```

然而，FORTRAN 语言并不是惟一的罪人。发明 FORTRAN 语言时，还没有计算机语言的文法定义；当时仅有自然语言的有关研究刚认识到上下文无关语言的概念。但 FORTRAN 语言的设计师之一 John Backus（约翰·巴克斯）接着改造了这一新概念，其成果现在以他的名字命

名：Backus-Naur Form (BNF, 有时也称为 Backus Normal Form (巴克斯范式))。BNF 是一种为程序设计语言书写文法的方法，首先应用在 ALGOL 60 语言的规格说明中。本书中使用的文法与 BNF 有少许区别，但这仅仅是表面上的区别。在 BNF 中，上述英语语言的文法片段如下例所示：

```

<sentence>      ::= <noun-phrase> <verb> <noun-phrase>
<verb>          ::= hit
<noun-phrase>   ::= <article> <noun>
                 ::= <proper-name>
<article>       ::= a | the
<noun>          ::= ball | bat
<proper-name>   ::= Peter

```

Niklaus Wirth (尼克劳斯·沃思) 试图简化 ALGOL 语言编译程序的设计方面，同时保留它的一些形式化性质，其结果是 1970 年诞生了 Pascal 语言。Pascal 语言虽经形式化文法精心设计，但仍潜伏了一个错误，导致文法是二义的。该问题源自对 **if** 语句中可选 **else** 子句的解析，因为当并非所有嵌套的 **if** 语句都有一个 **else** 子句时，其文法无法形式化地指定如何将一个 **else** 关联到嵌套的 **if** 语句。沃思意识到这一问题，并提出一种特别的解决方案(**else** 与内层的 **if** 配对)，而不是改正文法中的错误，其中有一个重要原因：没有二义性的语言用起来将更加笨拙。在实际程序中，大量的条件语句需要 **else** 子句，同时也有大量的条件语句不使用它；要求在所有情况下都带有 **else** 子句将使程序难以书写和阅读；但如果排除 **else** 子句又将削弱语言。因而 **else** 子句是可选的，而语言则是二义的。该语言的一个特点就是形式化的正确性让位于用户的便利。

沃思的下一代系统程序设计语言 Modula-2 以一种可读的且文法正确的方式解决了上述问题。Modula-2 语言保留了可选的 **else** 子句，同时又没有 Pascal 语言的二义性。但沃思似乎并没有真正地吸取教训，Modula-2 语言出现了另一问题：无法为其编译程序构造一个纯正的有穷状态扫描程序。以注释屏蔽部分代码（包括注释本身）的能力再次服务于程序员的便利，即使不惜以实现的简单性为代价。很不幸，这导致了定义有问题的（就算不是二义的）扫描程序文法。以沃思为榜样，大多数实现人员解决这一难题的办法就是忽略它，这种做法的后果是并非所有代码片段都能成功地通过在起、止位置添加注释分隔符被注释屏蔽。代码清单 1.1 展示了一个病态的例子，其中试图以注释屏蔽代码，在编译时不会出现错误，但却有意外的结果。

代码清单 1.1 在 Modula-2 语言中以注释屏蔽一些代码的失败尝试。注意，加下划线表示的新可执行语句其前身是一个字符串常量的一部分

<pre> MODULE pathological; (* 原程序 *) FROM InOut IMPORT WriteString; VAR anyString: ARRAY [0 .. 99] OF CHAR; BEGIN IF 1 < 2 THEN anyString := "Hello!" ELSE </pre>	<pre> MODULE pathological; (* 注释屏蔽前 2 条语句 *) FROM InOut IMPORT WriteString; VAR anyString: ARRAY [0 .. 99] OF CHAR; BEGIN (***** 新的注释区域开始 IF 1 < 2 THEN anyString := "Hello!" ELSE </pre>
---	--

```

anyString :=
  '*) WriteString("Surprise!"); (*'
END;  (* IF *)
WriteString(anyString);

WriteString(" What happened?")
END pathological.

```

```

anyString :=
  '*) WriteString("Surprise!"); (*'
END;  (* IF *)
WriteString(anyString);
新的注释区域结束 *****
WriteString(" What happened?")
END Pathological.

```

Hello! What happened?

Surprise! What happened?

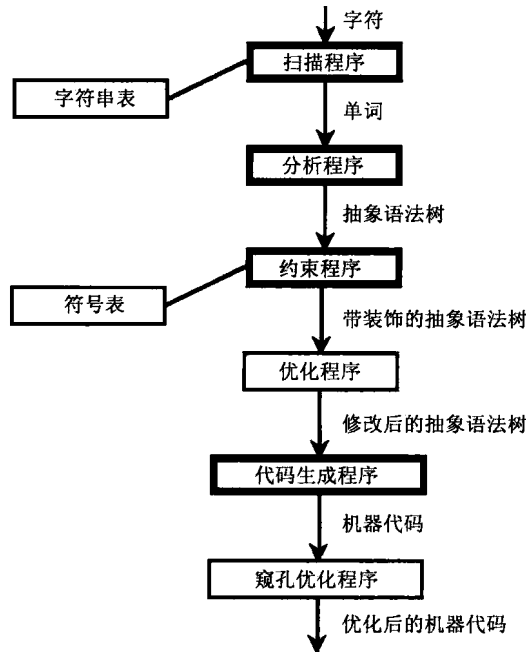


图 1-1 编译程序的组成部分

1.5 编译程序的结构

一个编译程序由四个核心部件组成，如图 1-1 中粗框所示：扫描程序、分析程序、约束程序以及代码生成程序。扫描程序读入源程序文本文件，将它看作由字符组成的串，并从中识别出一个由字和符号组成的流，这些字和符号称为单词（token）。分析程序将扫描程序输出的单词作为输入，识别出源语言的短语结构，并构造一棵抽象语法树（简称 AST）传递给下一步。约束程序强加了类型和声明规则的限制，并为 AST 添加“装饰”。约束程序通常被视为分析程序的一部分，因为它们的工作是如此紧密地结合在一起。代码生成程序根据抽象语法树构造目标机器代码。扫描程序和约束程序通常各自构建私有的数据结构，用于辅助各自的识别过程；扫描程序建立一个字符串表，其中存放了标识符的拼写、字符串常量等；约束程序则建立一个符号表，其中包含标识符的语义信息，这些信息是在源程序中声明或使用标识符时收集的。

除了上述四个核心部件之外，编译程序还可包括一个或多个优化模块。有两类优化，它们对目标机器代码特性的依赖程度有很大区别。机器无关的优化通常在抽象语法树上操作，将抽

象语法树整形以便产生更高效的代码，而不必考虑目标机器代码的特性。窥孔（**peephole**）优化是机器有关优化程序的最常见形式，在代码生成程序产生的目标机器代码上操作，对这些代码进行某些局部的改进。“窥孔优化”的名字缘于这一优化形式每次仅检查少量指令（有如通过锁眼之类的有限视角进行窥视），也只修改所观测到的指令。

尽管我们将扫描程序、分析程序、约束程序以及代码生成程序展示成一排，好像工厂的组装线上的一个个岗位，但编译程序的实际结构却往往因大多数计算机的工作方式而有所不同。如图 1-1 所示，扫描程序、分析程序、约束程序、优化程序以及代码生成程序看起来是相互独立的程序，每一程序读入一个输入文件，进行一些内部加工后，再写入一个输出文件，供下一程序读入。而真正的编译程序往往由分析程序作为一个主程序，由它调用扫描程序、约束程序以及代码生成程序等子程序，如图 1-2 所示。对扫描程序的每次调用仅返回单个单词；对约束程序的每次调用都是在符号表中查找单个标识符，或校验单个表达式运算符的操作数类型；对代码生成程序的每次调用均为发送单个语义动作，产生一条或多条目标代码指令。

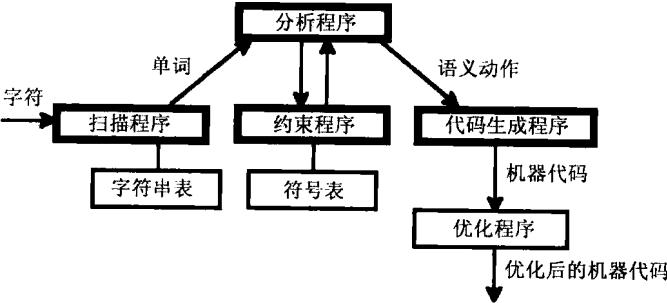


图 1-2 编译程序的结构

1.5.1 词法分析

虽然语言手册在定义一门程序设计语言时通常采用单一文法，一路往下定义，直至一个个字符组成文法的各个部分，然而将字符识别过程与结构识别相分离往往效率更高，这要求分别定义语言的词汇文法和短语结构文法。由于理论上的原因，分析程序无法匹配一个声明与引用必须拼写相同的标识符，因而将标识符（以及各类常量）的拼写从分析程序执行的短语结构识别中分离出来也会带来方便。一个称为扫描程序的模块负责完成此项任务。

扫描程序的规格说明由一个文法给出，该文法定义了源语言中所有合法的单词。一个单词表示分析程序期待读入的单个原子符号，例如任意的特殊字符、数字、字或由字符组成的一个串。“单词”这一术语指的是任一文法所定义的语言中的一个符号，尽管我们通常仅限于在分析程序的文法中使用该术语。扫描程序是一个分析输入文本文件并识别其中合法单词的过程，这些单词即是由字符组成的串（每个串都属于词汇文法定义的语言），将作为一个个单词传递给分析程序。非法的单词也会被识别出来，并采取合适的出错恢复动作使编译过程得以继续；如果一个编译程序能尝试尽可能多地分析源程序，而不是遇到第一个错误就停下来，那么这样的编译程序更加实用。

扫描程序的功能称为词法分析，因为它分析输入语言的词素（即一个字）。每一识别出来的词素被转换为扫描程序输出流中的一个单词。通常分析程序调用扫描程序以获得一个单词，扫描程序一旦识别到一个单词就立即返回给分析程序。因而，单词流并不是真正汇集到一个文

件或其他数据结构中，我们只是为了便于区分编译程序的各组成部分才如此看待它；在任一时刻，仅存在一个这样的单词，它就是分析程序正在处理的下一单词。

典型的程序设计语言可能定义了 50~100 个不同的单词；在源文件中，每个单词通常表示为若干字符组成的串。此外，源程序中可能含有许多空白（空格、换行、制表等字符）以提高可读性，以及对程序的含义不起作用的注释（从计算机的角度看）；扫描程序将单词传递给分析程序之前，会将它们全部删除，其结果是扫描程序产生的单词流与输入文本文件相比，有一个数量级甚至更多的缩减。编译的大部分时间往往花在扫描程序上，因而极大地提高扫描程序的效率相当重要。将扫描程序从分析程序中分离出来的原因之一，是因为识别单词的词汇文法更简单，并且可以用比分析程序所需的那类文法更为高效的方式处理。

在现代程序设计语言中，单词可看作是输入程序中粒度仅大于单个字符的部分。图 1-3 展示了从字符到单词、再到抽象语法树的这一层次。

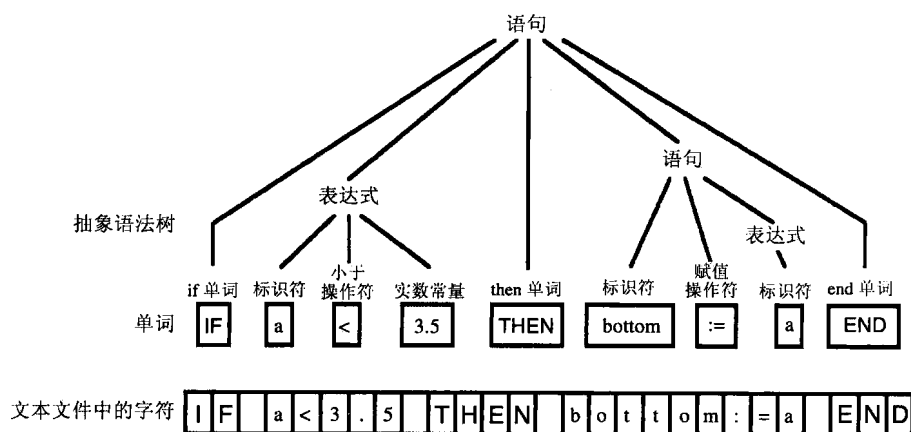


图 1-3 字符、单词和抽象语法树示例

1.5.2 字符串表

扫描程序传递给分析程序的单词已清除了所有附加的可辨识特性，例如标识符是如何拼写的、常量的值等等；分析程序的文法并不关心这些细节。然而，这些信息不能像注释或单词之间多余的空白那样弃如敝屣。实际上，编译程序的一个功能正是将特定的内存位置与每一个唯一的变量标识符相关联，并在一个过程或程序中始终如一地处理它们；因而标识符的拼写是有意义的，而且必须加以检查。

FORTRAN 语言最初版本的编译程序规定，所有标识符不可长于一个计算机字所能容纳的字符数；但现代程序设计语言力图支持相对较长的标识符。即便很少标识符会达到最大长度，在符号表中仍需为最大字符数分配存储空间；除了由此带来的低效之外，每次在符号表中查找名字时，比较两个长字符串的相等也是笨拙和费时的。

解决上述两个问题的一种方法是将每一个惟一标识符的拼写编码为单个整数或指针，该值可在有需要时用于访问标识符的拼写（通常仅用于出错信息和装载映像表），并且比较一对单个原子值的相等远快于字符串的比较。标识符的拼写压缩存储在一个字符串表中，指向字符串表的指针或下标可作为一个标识符的现成的惟一引用。扫描程序的任务之一就是管理字符串表，插入新的标识符和其他字符串，查找其中已有的字符串并将其惟一下标传递给约束程序。

1.5.3 语法分析

分析程序是编译程序的驱动引擎，控制着扫描程序和代码生成程序。它识别以单词流表示的输入程序的短语结构（即语法）。我们一般认为分析程序的输出是一个抽象语法树，但往往并不建立这样的树，而是由分析程序的执行流追踪遍历这棵抽象树或想象树的一条路径，以某种指定的方式访问其中的每一结点。因而，如果我们以图形方式记录下执行轨迹，在编译程序结束时它就是一棵抽象语法树；但在任一时刻，仅存在抽象语法树的一个片段（可能是树中从根结点到当前结点的路径），并且难以明确地标识出来。

在理论上，抽象语法树相当于学过高中英语的学生所熟悉的语法图：每个单词由树中的某一部分表示，整个结构由各部分的相互连接来定义。

有两类常用的确定分析算法。最古老的是自顶向下方法，从顶部（大多数数学上的树都是从顶部画根结点）出发，自根结点向下构造抽象语法树；对于根据文法手工编程而言，这种方法也是相当简单的。自底向上的分析从单词开始，逐个分枝构造这棵树，一边识别更多的程序，一边将分枝连接为越来越大的子树，直至所有子树最终连接为一个表示整个程序的抽象语法树。自顶向下和自底向上分析都是按从左到右的次序读入源单词，但是应用文法规则的次序则不相同。据此，自顶向下分析有一个等价的术语：LL 分析（第一个 L 表示从左到右扫描，第二个 L 表示最左推导）；自底向上分析也有一个等价术语：LR 分析（L 表示从左到右扫描，R 表示逆序的最右推导）。

1.5.4 约束

约束程序即编译程序中的静态语义处理阶段，通常被视作分析程序的一部分，但实际上其实现方式与分析程序的语法检查功能完全不同。语义分析程序这一术语常用于表示一个与某一部分代码生成功能相结合的约束程序，但我们宁可区别对待这些功能。约束程序是辨别错误源程序与合法源程序时必不可少的部分，但不可能根据构造分析程序的同类文法自动地构造一个约束程序。

必须检查的约束包括校验所有的标识符在使用之前均已声明，过程调用中使用的标识符不是声明为枚举类型的常量，整数类型的表达式没有赋值给布尔类型的变量或与它们进行比较，等等。图 1-3 所示的程序片段中，约束程序将校验标识符 **a** 要么是一个实数类型的常量，要么是一个实数类型的变量，以及标识符 **bottom** 应声明为一个实数类型的变量。

本书在约束检查中使用属性文法。由于属性文法是用构造分析程序的上下文无关文法的自然扩展，这是实现约束的一种相当省事的方法。

1.5.5 符号表

符号表是由编译程序为被编译程序中各个标识符添加的语义信息组成的知识库，这些信息通常称为属性，但本书避免使用该术语，以防与属性文法混淆。符号表中的一个入口必须至少包含：某个引用，指向标识符的拼写（如果不是英文单词中的字符）；一个值，表示该标识符所命名的变量或过程的内存位置或访问路径；以及一些标志位，用于区别变量、过程和其他标识符。

1.5.6 代码生成

编译程序通常划分为两半。编译程序的前端注重分析，即识别一个合法的输入源程序文件；后端注重合成，即生成目标机器的代码。前端由扫描程序、分析程序和约束程序组成，后端则

由代码生成程序和各种优化模块组成。对一个解释程序而言，其后端则是产生计算结果的那一部分，尽管解释程序通常并不使用该术语。

编译程序设计人员的职责是定义一个特定的目标机器指令序列，该指令序列可正确地翻译抽象语法树中每一可标识的结点。有文献曾报导过几项研究，它们试图给出目标机器语言的文法和语义规格说明，从而可构造一个合适的“编译程序—编译程序”推导出必要的代码生成规则，但这些成果在实际应用中并不十分成功。本书采取更传统的立场，要求编译程序设计人员自己给出这些变换的规格说明。

代码生成程序是编译程序的一部分，它负责根据源程序的内部表示真正地生成输出代码。在“一遍”编译程序（正如本书作为练习的许多实验项目）中，代码是根据分析程序文法中的语义动作规格说明“即时”生成的：只要被识别的一个源程序结构可产生目标机器代码，就立即生成这些代码。这种做法限制了所能编译的语言类别，以及所生成代码的效率。

大多数编译程序的目标语言是目标机器的本地机器语言。最早的编译程序生成汇编语言并放在一个文本文件中，然后将该文件转交一个汇编程序处理。这种方案在简单的“一遍”编译程序中会带来一些好处，但额外的汇编步骤所带来的低效很容易抵消这些好处。虽然本书有少量练习题涉及源程序到源程序的翻译问题，即将一种高级程序设计语言翻译成另一种高级程序设计语言，但练习的重点仍是直接生成机器语言（也称为本地代码）。

计算机硬件通常依照不同的性能和体系结构规格说明进行设计，其中某些硬件对同一源程序必须生成的代码类别会造成深刻影响。本书不打算详细讨论所有常见的体系结构，而是深入研究一种零地址栈机器，因为这是生成代码的最简单目标机器。代码优化涉及对寄存器的体系结构以及内存到内存的操作进行研究。我们相信，在代码生成基本概念方面打下的扎实基础同样适用于几乎所有的体系结构，不管特定的操作细节是否相同。勤奋的读者可找到大量机会为不同的目标体系结构生成代码。

1.5.7 优化

优化是编译程序为改进执行时间或代码空间而修改其内部数据结构或所生成代码的过程。尽管最早的编译程序已可执行许多优化，但此后的优化算法研究仍做了大量工作。某些常用的算法基于坚实的理论，但在实际应用中许多优化策略的结果却是特设的代码。在优化阶段，很少见到像将编译程序前端的设计简化为一个带属性的上下文无关文法的机械翻译这种自动生成方式；换言之，编写一个正确的优化程序远比编写一个正确的分析程序复杂。

在编译程序设计中，不同的代码生成策略会带来或多或少的优化机会。如果一个编译程序中的代码生成程序足够完善，它可能已经隐式地完成了一些优化任务；而对于那些只有非常简单的代码生成程序的编译程序，这些任务将留待该编译程序的优化程序完成。因而，一个追踪寄存器内容的代码生成程序可消除冗余的装入指令，否则这一优化工作将留待窥孔优化完成。另一方面，将特定的优化变换应用到程序的抽象语法树可能会带来另外的优化机会，否则不可能发现这些机会。例如，将过程体回代到过程的调用点后，如果遇到编译时可求值的常量参数，则可执行常量表达式求值和消除无用代码等优化。

本书大部分关于编译程序构造的练习都以微型计算机作为目标机器，这些计算机缺少丰富多彩的指令形式支持大多数机器有关的优化变换。尽管如此，大多数经典的机器有关优化算法仍是有意义的。

构造一个实用的编译程序时，通常根据文法自动地构建扫描程序和分析程序；属性文法的最新研究进展也开始为约束程序带来自动化，但代码生成和优化仍需花费大量的手工编程功夫。相应地，构造一个完整编译程序的开销大部分集中在代码生成和优化阶段，以及约束程序（取决于该阶段未自动化的程度）。如果一个编译程序只有很少或甚至没有优化功能，或者一门语言仅有为数不多的约束需求，那么这样的编译程序和语言更容易实现，因而在市场上往往更有优势。一个新编译程序的实现人员可能期望在约束程序上花费的精力与在扫描程序和分析程序加在一起上所花费的精力一样多。一个类似于本书编译程序练习的简单代码生成程序并不会比一个约束程序更难，但一个合理的优化程序所需花费的精力往往多于编译程序所有其他部分加在一起所需花费的精力。

符号

→ 在文法重写规则中读作“重写为”，例如：

vowel → "a" | "e" | "i" | "o" | "u"

表示一个名为 vowel 的语法形式可重写为字母 a、e、i、o 或 u 之一。

| 在文法重写规则中读作“或”，表示重写的有两个选项。

::= 箭头“→”在 BNF 中的等价形式。

<> BNF 中用于标识一个语法名字的符号，以区别语言中的单词。例如：

<vowel> ::= a | e | i | o | u

本书将单词用引号括起来，而用普通标识符作为语法名字，因为这样更一致并更易于自动化。

缩略词

AST Abstract Syntax Tree，抽象语法树，是分析程序的隐含输出。

BNF Backus-Naur Form 或 Backus Normal Form，巴克斯范式，是一种书写文法的形式。

HLL Higher-Level Language，高级语言，例如 FORTRAN 和 Modula-2 语言。

LL 从左到右扫描，展开最左的非终结符，是一种简单的分析策略。

LR 从左到右扫描，逆序展开最右的非终结符，是一种更强大的分析策略。

关键词语

code generator（代码生成程序） 基于编译前端产生的信息（最常见的是单独的语义动作或一个抽象语法树）为目标机器产生代码的编译过程。

compiler（编译程序） 是一个将输入源程序（通常是一个文本文件）翻译为输出目标程序（通常是某种机器语言的“目标模块”）的程序。

back end（后端） 生成目标机器的代码。后端的组成部分包括代码生成程序，以及可有可无的一个或多个优化程序。

front end（前端） 识别合法的输入程序文件，并分析其结构。前端由一个扫描程序、一个分析程序以及一个约束程序组成。

constrainer（约束程序） 编译程序中一个过程，负责在源程序中强加关于类型与声明的规则。

grammar（文法） 书写语言中合法句子的一系列规则。

lexical（词汇文法） 定义一门计算机语言中单词的正确形式。

phrase structure（短语结构文法） 定义一门计算机语言中的单词如何组装在一起，形成一个语法正确的程序。

HLL (高级语言 (Higher-Level Language) 的缩写) 其表示法比目标机器语言更接近于待求解的问题。
input program (输入程序) 一个文本文件或字符流。

interpreter (解释程序) 一个执行输入源程序的程序, 不会将它翻译为任何特定的机器语言输出。

lexeme (词素) 指源程序语言中的一个单词或字, 由一个字符或一个字符序列组成。

lexical analysis (词法分析) 负责识别并标识一门语言中词素 (即一个字) 的扫描程序动作。

parser (分析程序) 是编译程序的一部分, 根据一门计算机语言的短语结构分析该语言中的一个句子 (程序)。

parsing ((语法) 分析) 分析程序执行的动作, 识别一个输入程序的短语结构或语法。

bottom-up (自底向上分析) 构造一棵抽象语法树时, 从叶结点的单词出发, 逐一建立抽象语法树分枝, 直至到达根结点。LR 是一种自底向上的分析策略。

LL (LL 分析) 意即从左到右扫描一个输入串, 基于最左的非终结符判断选用哪一个文法产生式 (称为最左推导)。

LR (LR 分析) 意即从左到右扫描一个输入流, 展开最右的非终结符, 其逆序是一个最右推导。

top-down (自顶向下分析) 构造一棵抽象语法树时, 从顶部的根结点出发向下构建, 直至到达单词叶结点。LL 是一种自顶向下的分析策略。

scanner (扫描程序) 是从作为输入源程序文本的字符串中识别合法单词的编译程序过程。

string table (字符串表) 扫描程序使用该数据结构存放标识符和字符串常量的拼写。

symbol table (符号表) 约束程序使用该数据结构存放源程序文本中标识符的语义信息。

token (单词) 由扫描程序识别的单个原子符号。一个单词可以是一个数字, 或一个字符串常量, 或诸如 “;” 等标点符号, 或诸如 “IF” 等保留字 (或关键字), 或诸如 “:=” 等多个字符的符号, 或标识符 (例如过程、变量或常量的名字)。

translator (翻译程序) 一个将源程序文本翻译为目标 (机器) 语言的程序。

assembler (汇编程序) 将低级的符号化代码翻译为机器代码。

compiler (编译程序) 将高级语言的源程序翻译为以后再执行的目标机器代码。

interpreter (解释程序) 立即执行源程序文本, 不会将源程序文本翻译为任何机器代码。

tree (树) 一个层次化的数据结构 (一个有向无环图, 且根结点到任何其他结点仅有一条路径)。

AST (抽象语法树 (Abstract Syntax Tree) 的缩写) 抽象语法树由分析程序构造, 其中的内部结点表示非终结符, 叶结点表示单词。

练习

1. 列出本章英语语言的小文法所产生的全部句子, 并展示如何从初始字 “sentence” 推导出各个句子。
2. 修改该英语语言的文法, 使得一个专有名词不可同时出现在动词的两边。

复习小测验

指出下列陈述是否正确:

1. 扫描程序执行词法分析。
2. 汇编语言是一种高级语言。
3. 分析程序先构造一个字符串表, 然后再用它构造一个抽象语法树。
4. 分析程序识别一个输入流的短语结构。
5. 解释程序是一种运行较慢的编译程序。
6. 属性文法定义了如何使用符号表。

编译程序实验项目

1. 列出 Modula-2 子集 Itty Bitty Modula 语言中所有可能的单词；列单词时仅限于 **IF** 和 **WHILE** 控制结构以及整数和布尔类型操作，不必考虑集合、常量标识符或独立的模块。注意，你应假定所有标识符表示为单个通用的“identifier”单词；必要时请参阅附录 A 中 Itty Bitty Modula 语言的语法图。
2. 列出组成上表中全部单词所需的 ASCII 字符。
3. 思索 Itty Bitty Modula 语言的编译程序应强加哪些关于标识符和表达式的约束。注意单词次序和短语结构是分析程序处理的语法问题，而不是约束程序处理的语义问题。

进一步阅读

Allen, J. *Natural Language Understanding*. Reading, MA: Benjamin/Cummings, 1987.

参阅第 17.4 节：“文法与生成工具”。

Dowty, D.R., Karttunen, L., & Zwicky, A. (ed.), *Natural Language Parsing*. New York: Cambridge University Press, 1985.

Garzdar, G. "Phrase Structure Grammar." In *The Nature of Syntactic Representation*, ed. Jacobson, P. & Pullman, G.K. Dordrecht: D. Reidel, 1982, pp.131-186.

Wirth, N. "From Modula to Oberon." *Software – Practice and Experience*. Vol.18, No.7 (July 1988), pp.661-670.

第2章 文法：乔姆斯基层次

本章旨在：

- 理解文法的结构
- 综述乔姆斯基层次
- 区分上下文敏感文法、上下文无关文法和正则文法
- 研究分析树的结构
- 揭示文法及其机器之间的关系
- 介绍规范推导
- 揭示有穷状态自动机的局限性
- 研究文法中的计数方法
- 文法思维艺术的实践

2.1 简介

本章将揭示本书的核心主题：文法。由于编译程序前端在很大程度上可根据定义一门语言的文法自动地构造，所以将我们对这些内容的理解建立在严格数学理论的基础上显得至关重要。文法有一个特征与它所产生语言的复杂性有关。根据 Noam Chomsky（诺姆·乔姆斯基）对自然语言的研究，可找出四个级别的语言复杂性，通常称之为乔姆斯基层次。每一语言级别均由一类自动机标识，这类自动机可识别该层次中任一语言的串。我们对乔姆斯基层次中两个较高的（更严格的）级别特别感兴趣：正则语言和上下文无关语言，因为可从这两类语言的文法自动地构造高效的扫描程序和分析程序。

2.2 文法

文法在数学上定义为一个四元组，即它由四个不同的部分组成：字母表、非终结符、产生式以及一个目标符号。它们在本书中分别采用符号 Σ 、 N 、 P 和 S 表示，再用圆括号括住并用逗号分隔：

(Σ, N, P, S)

前三个符号均表示集合，第四个符号指定第二个集合中的一个特定元素。因而，一个文法由三个数学上的集合以及其中一个集合中的某个特定元素组成。

2.2.1 字母表与串

第一个集合 Σ 是字母表，即终结符的集合，是一个由可用于形成语言中的句子的所有输入字符或符号组成的有穷集。通常认为英语的字母表是从 A 到 Z 的 26 个字母，但在我们的定义中还须包括标点符号以及字母之间的空格，因为它们都是正确英语句子的重要组成部分。

大多数程序设计语言都被编码为文本字符组成的串，因而它们的字母表就是文本字符的集合，通常是定义明确的某一计算机集合，例如 ASCII（读作 ask-key，表示 American Standard Code for Information Interchange，意即美国信息交换标准编码）。

通常采用两个文法定义一个编译程序。扫描程序文法的字母表是 ASCII 或 ASCII 的某一子

集：而分析程序文法的字母表则是由扫描程序生成的单词组成的集合，完全不是 ASCII。我们抽出或删除标识符单词的各个拼写，即扫描程序每次识别并报告的标识符是一个通用的“identifier”单词，因而单词的集合仍是有穷的，实际上它往往比扫描程序的字母表更小。

本书的许多例子和练习重点关注文法的具体方面，因而我们经常使用非常有限的字母表，这些字母表仅由罗马字母中的前几个小写字母组成，并且显式地定义，例如 $\{a, b, c, d\}$ 。在一门真实语言的文法中，我们通常将字母表表示为一个个像字符常量一样用引号括住的字符，例如实数常量的字母表为：

$\{ '+', '-', '.', 'E', 'O', \dots, '9' \}$

字母表中的终结符可根据文法规则组装成任意长度的串。术语“串”用于表示按任一明确次序排列的 0 个或多个终结符组成的序列。尽管我们不考虑无穷的串，一个串却可以是任意长的；因而任一给定的（有穷）字母表，均有无穷数目的可能的串。一个文法可定义某一特定的仅包含有穷数目的串（甚至可能仅有一个串）的语言，但值得关注的程序设计语言都是无穷的，至少在实践中如此。

令 $\Sigma = \{a, b, c, d\}$ ，考虑一个关于串的例子。 Σ 中终结符的可能串包括 *aaa*、*aabbccdd*、*d*、*cba*、*abab*、*ccccccccccaccccc* 等。空串通常记为 ϵ 。 Σ 的所有可能的串组成的集合，包括空串 ϵ ，记为 Σ^* （读作 sigma star）。 Σ^* 称为字母表的闭包，表示由字母表中符号组成的任意串，包括空串。这个星号称为克林星号，因逻辑学家 Stephen Cole Kleene（史蒂芬·戈尔·克林）而命名。本书在多个不同的上下文中使用了克林星号，表示 0 个或多个所讨论的对象。在这里，它表示“字母表 Σ 中的 0 个或多个终结符组成的串”。一门语言是 Σ^* 的某一指定的子集。

2.2.2 非终结符与产生式

组成一个文法的数学定义的第二个集合是关于非终结符的集合，在四元组中用 N 表示，这是一个由不属于字母表的符号组成的有穷集。非终结符并不是串的集合，它只是一些符号，可看作表示或代表一个 Σ^* 子集的串集合。有一个特殊的非终结符称为目标符号，它恰好表示语言中所有的串。非终结符亦称语法范畴或文法变量。在本书的例子和练习中，通常使用罗马字母中的前几个大写字母表示非终结符，以 A （有时以 S ）表示目标符号。在真实程序设计语言的文法中，更喜欢用一个暗示了非终结符含义的标识符作为非终结符的符号。终结符与非终结符合在一起的集合，称为文法的词汇表。

一个文法的产生式（上述四元组中的 P ）是一个重写规则的集合，每一规则均写成由一个箭头分隔的两个符号串。箭头左右的符号均可从终结符或非终结符中抽取，并受到文法形式方面的某些限制。越简单的语言可由越简单（即受到越多限制）的文法描述。从定义语言的产生式的形式看，越复杂的语言要求有越少的限制。

2.2.3 若干文法例子

考虑一个非常简单的语言，它可由所有可能的文法中最简单的文法定义，该文法的形式受到最严格的限制。以下是一个完整的文法，它定义了包含字母 a 、 b 和 c 中两个字母（可能是相同的字母）以任意次序组成的所有串：

$G_1 = (\{a, b, c\}, \{A, B\}, \{A \rightarrow aB, A \rightarrow bB, A \rightarrow cB, B \rightarrow a, B \rightarrow b, B \rightarrow c\}, A)$

该文法中，字母表 Σ 是小写字母的集合 $\{a, b, c\}$ ；非终结符是大写字母的集合 $\{A, B\}$ ，其中字母 A 是目标符号；产生式的集合是如下列表：

$$\begin{array}{ll} A \rightarrow a B & B \rightarrow a \\ A \rightarrow b B & B \rightarrow b \\ A \rightarrow c B & B \rightarrow c \end{array}$$

非终结符 A 表示“语言中的所有串”，非终结符 B 表示“由一个符号组成的所有串”。从目标符号 A 出发，可应用产生式规则重写它（或其中的一部分），直至到达一个终结符组成的串。我们总是从目标符号出发，上述文法中即是 A （目标符号通常也称为起始符号，因为我们从这一符号开始）：

A

前三条规则（即产生式）中的任一条都可应用于这一初始串，我们随意挑选了第二条，将 A 替换为 bB ：

bB

只要有可能，我们就继续应用规则。由于不再有 A 可以重写，前三条规则不可以再应用；但后三条规则中的任一条都可以，因为其中的任一条规则都可替换待处理串中的 B 。我们又随意挑选了最后一条规则并重写上述串：

bc

此时串中仅含终结符，不再有非终结符，因而重写过程终止。

这一系列的步骤称为推导，其结果是推导出该语言中的一个串。上述推导还可写成如下的单行形式：

$$A \Rightarrow bB \Rightarrow bc$$

双箭头读作“推导”，表示推导中的一个步骤。因而， A 推导出串 bB ， bB 再推导出终结符串 bc 。只要待处理的串中仍有非终结符，推导过程就不会终止；如果只剩下终结符，则终止重写规则的应用。这也正是这些符号被分别称为“非终结符”和“终结符”的原因。在推导过程中的任一步，均可选择其左部在串中某处出现的任一规则，并使用相同规则的右部替换这一左部。

有时使用克林星号作为推导的简写会更方便：

$$A \Rightarrow^* bc$$

带星号的双箭头表示非终结符 A 可用 0 个或多个推导步骤推导出串 bc 。

选择不同的产生式可推导出该语言中的另一个串：

$$A \Rightarrow aB \Rightarrow aa$$

一路作出不同的选择，容易看出该文法可产生的全部串是 9 个双字母的串，它就是我们定义的语言。文法 G_1 是用于定义扫描程序这类文法的一个例子，尽管这里的语言是有穷的，而大多数扫描程序文法定义的语言是无穷的。

另一个例子是文法 $G_2 = \{ \Sigma, N, P, E \}$ ，其中字母表 Σ 是集合 $\{ n, +, *,), (\}$ ； N 是非终结符的集合 $\{ E, T, F \}$ ； P 是如下产生式的集合（添加了编号，以便于稍后引用）：

$$\begin{array}{ll} D.1 & E \rightarrow E + T \\ D.2 & E \rightarrow T \\ D.3 & T \rightarrow T * F \\ D.4 & T \rightarrow F \\ D.5 & F \rightarrow (E) \end{array}$$

D.6 $F \rightarrow n$

第一条产生式的左部是单个非终结符 E ，其右部是一个由终结符“+”分隔的两个非终结符 E 和 T 组成的串。该产生式的含义是，每当该文法用于生成它所定义的语言中的一个串时，只要待处理的串中出现了非终结符 E ，该非终结符就可被重写为串“ $E + T$ ”，而待处理串的其余部分保持不变。从目标符号 E 出发，可应用第一条产生式将它重写为“ $E + T$ ”，然后继续应用该产生式重写待处理串左端的 E ，形成一个新的待处理串“ $E + T + T$ ”，如此类推。

$$E \Rightarrow E + T \Rightarrow E + T + T \Rightarrow \dots$$

使用文法 G_2 可从目标符号 E 推导出串 $n + n * n$ ，如下所示：

E	目标符号
$E + T$	应用规则 D.1
$E + T * F$	应用规则 D.3
$T + T * F$	应用规则 D.2
$F + T * F$	对最左的 T 应用规则 D.4
$F + F * F$	再次应用规则 D.4
$F + F * n$	对最右的 F 应用规则 D.6
$F + n * n$	再次对最右的 F 应用规则 D.6
$n + n * n$	又一次应用规则 D.6

我们也可将上述过程画成一棵树，称为推导树、分析树或抽象语法树，如图 2-1a 所示。以下写法：

$$E \Rightarrow^* n + n * n$$

表示非终结符 E 可用 0 个或多个步骤推导出串 $n + n * n$ 。上述推导有 9 行，其中的每一行均称为该语言的一个句型，最后一行称为该语言中的一个句子。语言中的一个句子是 Σ^* 中可从目标符号经过一步或多步推导出的任一串；一个句型 ω 是 $(\Sigma \cup N)^*$ 中（即包含任意数目的终结符和非终结符）满足以下关系的任一串：

$$S \Rightarrow^* \omega \Rightarrow^* \sigma$$

其中， S 是目标符号，而 σ 是 Σ^* 中的一个句子。换言之，一个句型是由终结符与非终结符组成的任一串，它可从目标符号推导出，并且可继续推导出语言中的一个句子。一个句子的分析树上的任何剪枝都是一个句型，只要该剪枝没有绕过下面的任何终结符。图 2-1b 展示了一个剪枝的例子，它标明了句型“ $F + F * F$ ”。注意，图中的灰色线条如果不是向上弯曲到终结符“+”，而是从它下面绕过，那么这就不是一个正确的剪枝：在该语言中，“ $FF * F$ ”并不是一个句型。

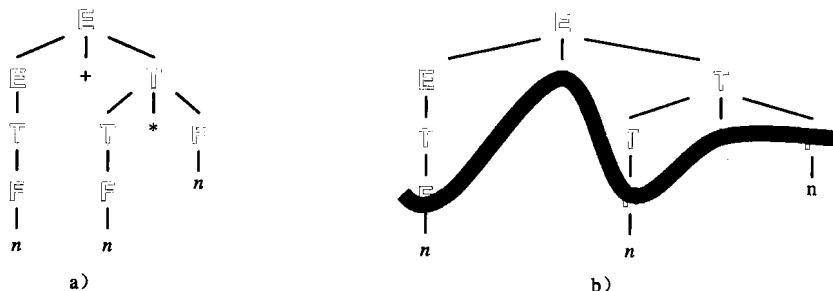


图 2-1 推导 $E \Rightarrow^* n + n * n$ 的分析树；图 b 中的灰色线条表示一个剪枝，展示了句型 $F + F * F$

机灵的读者会注意到，当该文法有多种选择时，应用产生式的次序不会带来任何差别。例如在 $n + n * n$ 的推导中，第二步可对非终结符 T 应用规则 D.3，也可对非终结符 E 应用规则 D.2：

$E \Rightarrow E + T \Rightarrow E + T * F \Rightarrow \dots$ 应用规则 D.3

$E \Rightarrow E + T \Rightarrow T + T \Rightarrow \dots$ 应用规则 D.2

不论采用何种次序，为推导出同一个的终结字符串，最终都会应用了全部相同的产生式。但指定次序时具有很大的自由度，这正是无二义的上下文无关文法的一个特点，本书稍后将深入讨论。

按从左到右次序阅读分析树的叶结点（分枝的顶梢），即形成该树所分析的句子，也称为其边缘。如果可为同一句子构造出两棵不同的分析树，则称该文法是二义的。

上述两个文法例子中，每一产生式的左端恰好只有一个非终结符，这一形式是用于构造编译程序的文法所特有的。然而还存在一类重要的文法不受此限制。第三个文法例子生成所有由相同数目 a 、 b 和 c 组成的、且按字母次序排列的非空串：

$G_3 = (\{a, b, c\}, \{A, B, C\}, P, A)$

其中， P 是如下产生式集合：

$A \rightarrow aABC$ $CB \rightarrow BC$

$A \rightarrow aBC$

$bC \rightarrow bc$ $aB \rightarrow ab$

$cC \rightarrow cc$ $bB \rightarrow bb$

以下推导可生成串 $aabbcc$ ：

A

$aABC$

$aaBCBC$

$aaBBCC$

$aabBCC$

$aabbbCC$

$aabbccC$

$aabbcc$

注意，该文法通过重写符号串（例如产生式 $aB \rightarrow ab$ ）保证了生成的串符合字母次序，尽管推导中有一步没有按字母次序排列。

2.3 乔姆斯基层次

上世纪 50 年代中期，语言学家诺姆·乔姆斯基定义了四个级别的文法，希望由此分析人类的自然语言。从自然语言的研究上看他并未成功，但“乔姆斯基层次”却很适合作为描述用于计算机程序设计的人工语言的形式化基础。

乔姆斯基定义了四个级别的语言复杂性，并将它们分别编号为 0 到 3（在计算机专业人士的优良传统中，编号方案总是从 0 开始）；此外还定义了四类文法，生成各自级别的语言。随后的语言学研究找出了四类对应的自动机，即抽象机器类型，它们恰好可识别相应文法所生成的语言中的串。表 2-1 给出了这四个级别。

表 2-1 语言和自动机的乔姆斯基层次

乔姆斯基语言类	文 法	识别器
3	正则文法	有穷状态自动机
2	上下文无关文法	下推自动机
1	上下文敏感文法	线性有界自动机
0	无限制文法	图灵机

2.4 文法及其机器

一个自动机的组成包括：一种控制机制，其中带有有穷数目的状态；一条某种形式的带，类似磁带播放机那样可读入和向前进，也可能可写入（记录）或双向移动。一个有穷的字母表定义了带上可能出现的符号，与一门语言的定义中的字母表相同。带上的初始内容是由字母表中的字符组成的串，即自动机的输入。从一个指定的起始状态出发，自动机基于带上找到的内容以及控制机制中的规则，依次经过它的各个状态，这些步骤称为变迁。任何时候自动机都可能停机，并且这表明自动机接受了该输入串；自动机也可能阻塞，即进入一个无任何规则可处理当前带上输入的状态，出现了阻塞情况即表明自动机拒绝该输入串。不同类的自动机主要通过它们的带的类别加以区别。

2.4.1 图灵机

乔姆斯基层次中限制最少的语言类别是 0 级，称为无限制文法；这类语言由各类自动机中最通用的图灵机识别。如图 2-2 所示，图灵机（通常简称为 TM）是上世纪 40 年代由 Alan Turing（艾伦·图灵）设计的一种计算装置的抽象模型，其中包含一个可在一条无穷的带上随处定位的读写头。图灵机的状态是一个有穷集，它的任何状态均须从起始状态出发，并且经过一系列变迁规则后进入这些状态。

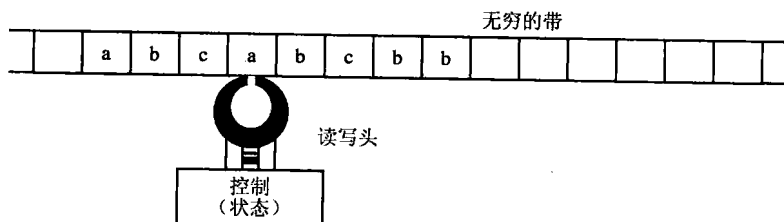


图 2-2 图灵机

对于单个状态和读写头下的单个字母表字符，每一变迁规则指定了三个动作：进入的下一状态、写回带上的一个新字符（也可能是同一字符）、将带移动一个字符位置的方向。读写带亦可留在原来的位置，并且任一规则也可指定在变迁结束后停机，表明图灵机接受这一输入作为它所识别的语言中的一个串。如果对于一个特定的状态和带上的符号没有变迁可用，图灵机将阻塞并拒绝这一输入串。0 级语言亦称递归可枚举语言，因为它是一个可由图灵机枚举（列举）的串组成的集合。

图灵机是许多可计算理论与计算复杂性理论的重要基础，但作为编译程序生产环境中的一个实用工具却是极其低效的。实际上已证明，对于任意输入带，我们无法判断一个图灵机是否终止（要么停机，要么阻塞）并产生结果。因而对于编译程序设计而言，0 级语言并不是有用

的语言类，因而本书不打算花费过多篇幅展开讨论。此处提及图灵机，是因为识别乔姆斯基层次中各语言级别的其他自动机均是图灵机的变形或受限形式。

2.4.2 线性有界自动机

乔姆斯基层次中的 1 级语言称为上下文敏感语言。上下文敏感语言可由一个包含一条有穷读写带的自动机正确地识别，这类自动机称为线性有界自动机（简称为 LBA），如图 2-3 所示。

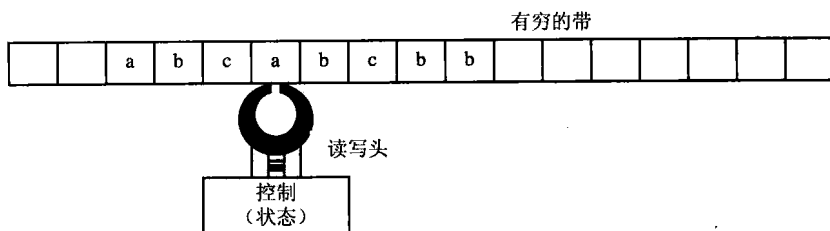


图 2-3 线性有界自动机

上下文敏感文法有两个限制：每一产生式的左部必须至少含有一个非终结符；产生式右部的符号不可少于左部，特别是不允许有空产生式。空产生式形如 $N \rightarrow \epsilon$ ，即产生式的左部是一个非终结符，而右部是一个空串。第二个限制有一个例外：如果产生式的左部仅由目标符号组成，且目标符号不出现在任何其他产生式的右部，那么右部允许是空串。如果要生成语言中的空串，这一例外是必不可少的。文法 G_3 是一个上下文敏感文法的实例。实际上文法 G_1 和 G_2 在形式上也是上下文敏感的，但它们有更严格的限制；我们通常用一门语言所属的最高乔姆斯基级别（即可生成该语言的受限最多的文法）描述该语言。

2.4.3 下推自动机

乔姆斯基层次中的 2 级语言称为上下文无关语言。上下文无关语言可由下推自动机（简称为 PDA）正确地识别，这类自动机只能读入其输入带，但还有一个可增长到任意深度的栈用于存放信息，如图 2-4 所示。一个包含一条只读带和两个独立栈的下推自动机等价于一个图灵机。

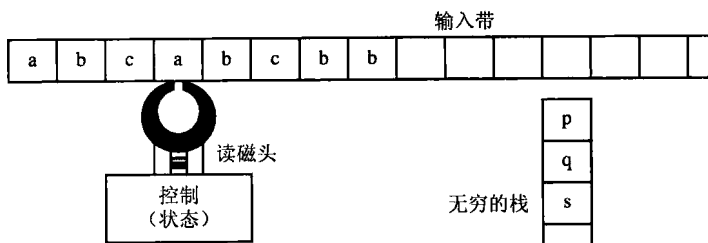


图 2-4 下推自动机

栈是一种仅允许从一端访问的线性数据结构，就像忙碌的行政人员的桌面上的纸堆，或餐厅中的自动盘碟伺服器。理论上栈可增长到任意深度，但实际上我们通常会基于计算机中的内存大小将栈限制在特定的大小。栈上有两个操作：压入和弹出，两者都在栈顶操作，如图 2-5 所示。压入操作将一个元素添加到栈顶，挡住原有的栈顶元素；弹出操作取走栈顶元素，露出下一元素供后续的弹出操作访问。

下推自动机的每一步都从栈中弹出一个符号，然后可立即将同一符号或其他符号压回栈中，或者也可不压入任何符号。下推自动机的每一步移动有可能读入其输入带并将输入带向前

移动，也可能停机（并接受输入）。由于下推自动机根据栈顶发现的符号以及读磁头下的符号决定状态的变迁，因此它可将栈作为读入信息的临时存储场所。

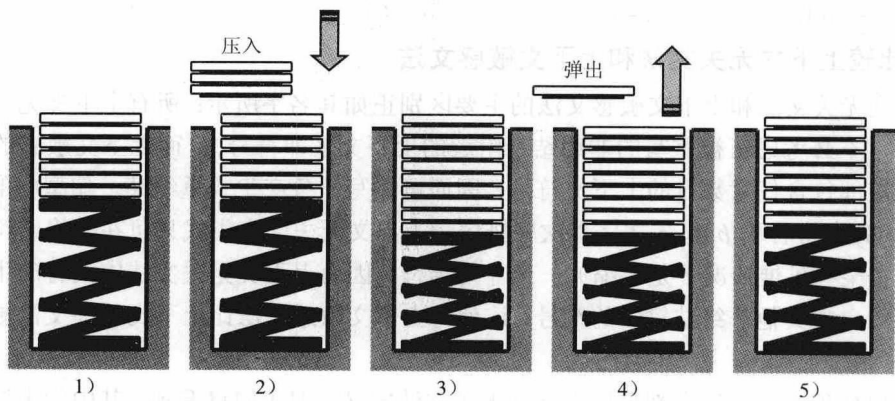


图 2-5 栈如何工作；图 2 将三个盘子压入盘子的栈中，图 4 弹出一个盘子

上下文无关文法比上下文敏感文法受更多限制，体现在每一产生式的左部仅允许出现单个非终结符（不允许含有终结符）。本书大量使用了“上下文无关文法”这一术语，因而将它简称为 CFG 会更方便。文法 G_2 是一个 CFG 的实例。与上下文敏感文法不同，在上下文无关文法中可放松对空产生式的要求，因为可证明这不会改变语言的基本能力。带空产生式的任一 CFG 可转换为最多只有单个空产生式的等价 CFG，这符合上下文敏感文法对空产生式的限制。本书第 7 章展示了如何从一大类上下文无关文法自动地构造确定的（线性时间的）下推自动机。

2.4.4 删除空产生式

利用以下转换，可删除上下文无关文法中的空产生式。对文法中的每一空产生式：

$A \rightarrow \epsilon$

找出并复制所有 A 出现在右部的产生式，并从得到的副本中删除 A。然后，若 A 不是目标符号，则删除这一空产生式；若 A 是目标符号且 A 还出现在某一产生式的右部，则选择一个新的非终结符 G 作为目标符号，并添加两条产生式：

$G \rightarrow A$

$G \rightarrow \epsilon$

接着可安全地删除原来的空产生式。

考虑如下的简单文法例子：

$A \rightarrow A a B \quad B \rightarrow B a$

$A \rightarrow \epsilon \quad B \rightarrow \epsilon$

对 B 的空产生式应用第一步后，添加了两个新产生式：

$A \rightarrow A a \quad B \rightarrow a$

再将第一步应用到 A 的空产生式，又添加了两个新产生式（其中一个来自原文法，另一个来自上一步添加的新产生式）：

$A \rightarrow a B \quad A \rightarrow a$

然后添加一个新的目标符号 G，得到的最终文法是一个没有多余空产生式的正确形式：

$G \rightarrow A \quad G \rightarrow \epsilon$

$A \rightarrow A a B$
 $A \rightarrow A a$
 $A \rightarrow a B$

$B \rightarrow B a$
 $B \rightarrow a$
 $A \rightarrow a$

2.4.5 比较上下文无关文法和上下文敏感文法

上下文无关文法和上下文敏感文法的主要区别正如其名字所示：所有上下文无关的产生式在应用时，不必考虑正被重写的非终结符附近的上下文（即符号）；而上下文敏感的产生式则可能在其左部包含任意数目的上下文符号，因而可编写一条产生式重组某一句型中的符号，正如在文法 G_3 中将若干 b 和 c 按字母次序排序。上下文无关的产生式只可在非终结符的原来位置展开（在某些扩展情况下是收缩）一个非终结符，因而上下文无关文法中没有产生式可影响到待处理的串中其他非终结符中的符号，这使得上下文敏感文法比上下文无关文法更强大且更难以识别。

所有现代程序设计语言都采用上下文无关文法定义，但我们将看到，其中的大多数在其语言定义中都嵌有上下文的敏感性。这种敏感性的一个例子是在 Pascal 和 Modula-2 等语言中，要求标识符在使用前必须先声明：这一声明的存在就是允许该标识符被使用的上下文。我们采纳的方案是经过精心约束的上下文无关文法扩展，而不是揭开上下文敏感性的潘朵拉之盒。这些扩展称为属性。因此，我们希望保留上下文无关文法带来的线性（快速）编译时间的所有好处，同时又不丢弃语言的特性，否则这些语言特性看起来似乎要求使用更低层的乔姆斯基级别。

2.4.6 有穷状态自动机

乔姆斯基层次中最高和受限制最多的级别是第 3 级的正则语言（有时亦称正则集），可由有穷状态自动机（简称 FSA；有时亦称有穷状态机，简称 FSM）识别，如图 2-6 所示。有穷状态自动机没有无穷的存储，仅允许在一遍中一次读入其输入。要记下关于输入带上一个符号的任何上下文信息，就必须将这些信息保留在机器的状态中。因为它是有穷的，因而机器状态中可存储的信息量是有穷的。正则文法是文法中受限制最多的，每一产生式的左部仅允许有一个符号（非终结符），右部仅允许有一个或两个符号；右部的第一个符号必须总是终结符，若有第二个符号则它总是非终结符。类似于上下文敏感文法和严格的上下文无关文法，正则文法也不允许空产生式，除非左部是目标符号的单条产生式，且目标符号不出现在任何产生式的右部。

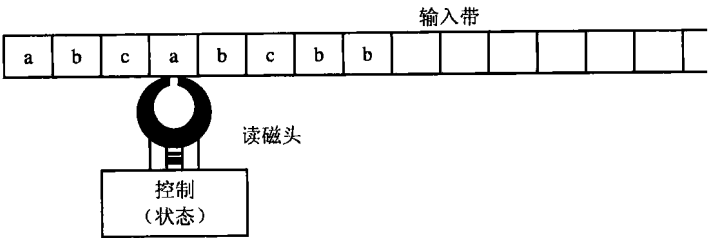


图 2-6 有穷状态自动机

正如上下文无关文法，我们可放松此规则而不改变语言的能力；但因为形式严格的正则文法与其等价的有穷状态自动机关系是如此直接，所以我们使用另一术语表达更宽松的文法：右线性文法是一个这样的文法，文法中每一产生式的左部恰好有一个非终结符，其右部有 0 个或多个终结符，后面跟着最多一个非终结符。在每一个右部含非终结符的产生式中，该非终结符

总是最右的符号，故其术语为右线性。若一个文法具有相同形式，但右部的非终结符在左端，则称左线性。本书第3章展示了正则文法、右线性文法、左线性文法之间是如何转换的，以及如何从它们构造等价的有穷状态自动机。

2.5 空串与空语言

考虑不同文法生成的语言时，很重要的一点是不要混淆一个生成空串 ϵ 作为其语言中的一个串的文法，与一个定义了空语言的文法（即该文法根本不生成任何的串）。文法 G_4 是一个仅定义了一个串（即空串）的文法：

$$G_4 = (\{a\}, \{A\}, \{A \rightarrow \epsilon\}, A)$$

文法 G_5 不生成任何串（即便是空串）：

$$G_5 = (\{a\}, \{A, B\}, \{A \rightarrow B, B \rightarrow aA\}, A)$$

乍看起来，文法 G_5 好像生成由 a 组成的串，但每一个半成品的串都包含一个非终结符。这一过程永远不会终止，因而它始终不会产生一个终结符的串。该语言是空的，因为这一文法不能生成任何仅有 0 个或多个终结符组成的串。

越是没那么简单的文法，越容易忽略那些不能产生任何串的产生式，例如文法 G_6 ：

$$G_6 = (\{a, b, c\}, \{A, B, C\}, P, A)$$

其中， P 是以下产生式的集合：

$$A \rightarrow BC$$

$$A \rightarrow aC$$

$$B \rightarrow bB$$

$$C \rightarrow cC$$

$$C \rightarrow a$$

尽管该文法有一条产生式看起来会产生由 b 组成的串，然而却无法摆脱该产生式中的非终结符 B ，因而这一非终结符及其产生式都是无用的。该文法只会生成两端各有一个 a 、中间由 c 组成的串。第一条产生式也是无用的，因为它永远不会应用在一个句子的推导过程。

2.6 规范推导

前面介绍了推导的概念，即如果一个非终结符 A 经过连续地应用产生式规则后，转换为一个由终结符和非终结符组成的串 ω ，则称 A 推导出 ω ，记为 $A \Rightarrow^* \omega$ 。再次考虑文法 G_2 ，其中 E 是目标符号；我们可以说在应用规则 $D.1$ 、 $D.2$ 和 $D.3$ 各一次，再应用规则 $D.4$ 两次，以及再应用规则 $D.6$ 三次之后， E 推导出该语言中的一个句子 $n + n * n$ 。这一过程可通过建立一棵分析树（又称推导树）以图形方式展示，如图 2-7 所示。

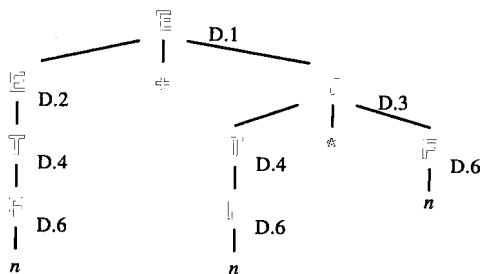


图 2-7 推导出文法 G_2 所定义语言中的串 $n + n * n$ 的分析树

由于这是一种二维图形表示，图 2-7 同时给出了产生式规则的全部 8 次应用；而计算机通

常是一种顺序装置，在顺序文件上操作，因而这种图形表示除直观解说之外就显得有些笨拙。然而，我们可通过每次仅应用一条规则将这一过程线性化，如此一来选择先应用哪条规则就有了多种可能。像这样一个无二义的上下文无关文法，对最终推导出来的同一个串而言，相同规则的应用次序对产生的分析树不会造成任何差别。

例如考虑图 2-8 所示的两种策略：一种称为最左推导，因为它总是对一个句型中最左的非终结符应用规则；另一种称为最右推导，因为它总是对最右的非终结符应用规则。这两种策略均推导出串 $n + n * n$ ，并且都构造了相同的分析树。只有以某一次序应用这些相同的规则，并构造这一相同的分析树，文法 G_2 才可能推导出这个串。

E		E	
E + T	D.1	E + T	D.1
T + T	D.2	E + T * F	D.3
F + T	D.4	E + T * n	D.6
n + T	D.6	E + F * n	D.4
n + T * F	D.3	E + n * n	D.6
n + F * F	D.4	T + n * n	D.2
n + n * F	D.6	F + n * n	D.4
n + n * n	D.6	n + n * n	D.6
a)		b)	

图 2-8 文法 G_2 中串 $n + n * n$ 的最左规范推导（图 a）和最右规范推导（图 b）

然而，应用规则的次序对于从目标符号 E 推导出句子 $n + n * n$ 的句型序列却是有意义的。图 2-8 中的两种推导称为规范推导，因为它们形成了两类分析程序构造方法的基础或规则（“规则”的拉丁文单词是 canon）。在手工编写的编译程序中，最左推导更容易实现。当从左到右读入一个输入串并采用最左推导时，从目标符号 E 开始自顶向下构造分析树；这样的编译程序称为自顶向下编译程序或 LL(k)编译程序。

最右推导可处理上下文无关文法的一个更大的、限制更少的子集。当从左到右读入一个输入串并采用逆序的最右推导时，从句子 $n + n * n$ 开始自底向上构造分析树；这样的编译程序称为自底向上编译程序或 LR(k)编译程序。尽管第 7 章讨论了自顶向下分析程序和自底向上分析程序的一些实现差异，本书大部分重点还是关注自顶向下分析程序。大多数分析程序自动生成工具是自底向上的，因为 LL(k)文法基本上比任何自底向上的技术都有更多的限制。

2.7 二义性

二义文法是指其语言中存在一个串有两棵不同分析树的文法。考虑文法 G_7 ：

- A.1 $E \rightarrow E + E$
- A.2 $E \rightarrow E * E$
- A.3 $E \rightarrow (E)$
- A.4 $E \rightarrow n$

它产生与文法 G_2 相同的语言。图 2-9 展示了同一个串 $n + n * n$ 的两棵不同分析树；通常很容易从产生式集合中推断文法的词汇表和目标符号，因而往往仅用产生式列表简写一个文法的规格说明。

文法 G_7 中串 $n + n * n$ 的两种分析需要相同的规则，但在自顶向下分析（最左推导）中应用这些规则的次序导致不同的分析树。如果将规则 A.1 先应用到目标符号 E，其结果是右图所示的树，其中加法运算符将第一个 n 与另外两个 n 之积相加；如果将规则 A.2 先应用到目标符

号 E，其结果是左图所示的树，其中乘法运算符将最后一个 n 与前两个 n 之和相乘。我们无法确切地定义有哪些特征导致一个文法是二义的，但可以证明某些文法是二义的。要证明一个文法是二义的，只需展示其语言中某一个串有两棵不同的分析树，但对一个特定的文法找出这样的例子可能并不会太容易。

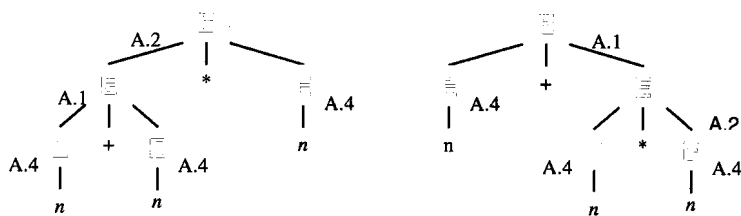


图 2-9 二义的文法 G_7 ，对同一输入串 $n + n * n$ 有两棵不同的分析树

程序设计语言 Pascal 有一条产生式导致它是二义的。作为一个独立的文法， $G_8 = (\Sigma, \{S\}, P, S)$ 定义了该语言的一个片段，其中 P 是下列产生式的集合：

```
S → "if" e "then" S "else" S
S → "if" e "then" S
S → p
```

且 $\Sigma = \{\text{"if", "then", "else", e, p}\}$ 。复合语句：

```
if e then if e then p else p
```

具有嵌套的 **if** 语句，但仅有一个 **if** 带有可选的 **else** 部分，这里的问题是：哪一个 **if** 带有 **else** 部分？Pascal 语言武断地定义了 **else** 属于最内层的 **if** 语句，但这一句子同样也很容易按另一方式进行分析。这导致该语句有不同的含义，如果 **else** 部分与内层 **if** 配对，则外层 **if** 的表示式求值为假时，不管内层表达式求值结果如何，内层 **if** 语句都不会执行；若要执行 **else** 部分，外层表达式必须为真，且内层表达式必须为假。另一方面，如果 **else** 部分与外层 **if** 配对，那么在任何情况下它仅依赖于外层表达式。

2.8 文法思维的艺术

本书以大量篇幅介绍了识别特定的上下文无关文法和正则文法所定义语言的计算机程序构造机制，这可以自动完成，即存在一个计算机程序根据文法定义构建一个识别器自动机。本书以较少的篇幅讨论如何编写一个文法，主要原因是我们已有构建自动机的算法，但不存在为任一语言需求编写一个正确文法的算法。这是一种艺术！然而，正如大多数艺术创作工作，我们可学习一些有助于设计优秀文法的已知技术和工具。

一种重要的工具是深刻理解不同乔姆斯基级别在描述语言特性时的局限和能力，重点是 1~3 级，0 级语言没有限制在语言中可定义什么或不可定义什么，因而无需进一步关注。乔姆斯基层次最高三级的主要区别是在生成的串中进行符号计数的灵活性；所谓“计数”，是指将一个集合（譬如该语言的一个串中的某类特定单词）中的元素与另一集合（可能是同一个串中的另一类单词）中的元素建立数学意义上的一一对应关系。像 Modula-2 或 Pascal 这类语言要求左圆括号与右圆括号必须配对，这时对左圆括号的计数就是将它们与数目相同的右圆括号配

对；由于圆括号可嵌套到任意深度，这种计数（在理论上）是无限的。

2.8.1 有穷状态自动机的局限性

有穷自动机无法对字母表中的任何符号进行计数或配对，除非是有穷数目。尽管大多数计算机其实就是有穷状态自动机（实际上没有一种计算机拥有无穷的栈或无限长的数据带），然而这种将一个系统视为海量状态的观点因系统的复杂性而变得没有什么价值，这就好像试图通过计算所有树上的树叶数量来测量一片森林。因而，我们仅考虑将有穷状态自动机和正则文法应用到只需要对少量元素进行计数的情况。

文法 G_1 是一个对少量元素进行计数的正则文法实例，它定义的语言由 Σ^* 中恰好两个字符的所有串组成。因而，该文法必须计数到 2。

另一个例子是为校验文字量的正确形式，对一个整数文字量中的位数进行计数。在由 0 个或多个数字（为简单起见，假设字母表中仅包含数字）组成的所有可能串的集合中，整型常量是其中那些至少含有一个数字的串；因而对生成这些文字量的文法，只需对第一个数字进行计数，长度为 0 的串不属于该语言，而所有其他的数字串均属于该语言，不管其长度如何。注意这里仅关注整数常量的形式，可以想像（实际上也很有可能）一个整数在形式上是正确的，但可能因太大而无法用目标机器的整型变量表示；我们将这一问题看作是一个语义问题，而不是扫描程序需处理的问题，也不是在正则文法中试图避免的问题（但请参阅练习 6）。

要保证一个串中的符号数目是奇数或偶数时，计数就没有那么显而易见。确定奇偶性只需计数到 2，因而正则文法就足够用了。以下是一个正则文法的产生式，该文法定义的语言是由偶数个 a 和奇数个 b 组成的所有串。

$A \rightarrow a B$	$C \rightarrow a D$
$A \rightarrow b C$	$C \rightarrow b A$
$A \rightarrow b$	$D \rightarrow a$
$B \rightarrow a A$	$D \rightarrow a C$
$B \rightarrow b D$	$D \rightarrow b B$

上述四个非终结符中， A 表示语言中的所有串，即由偶数个 a 和奇数个 b 组成的所有串。根据第一条产生式，该语言中一个串的可能组成是由一个 a 开头、后接一个由非终结符 B 表示的任意串。因而， B 表示由奇数个 a （即 A 中的偶数再减去开头的 a ）和奇数个 b 组成的所有串；类似地， C 表示由偶数个 a 和偶数个 b 组成的所有串。由于 0 是偶数，非终结符 A 亦可仅由一个 b 和 0 个 a 组成（ A 的第三条产生式）；同理，非终结符 D 由奇数个 a 和偶数个 b 的所有串组成，也可能仅由单个单词 a 组成。

一个有效的经验法则是，在正则文法中只要对一个符号进行计数，那么所需的非终结符数目就是计数必须达到的最大值；如果必须维护若干独立的计数，那么计数所需非终结符的总数就是所有各个需求的乘积。在上述例子中，希望对两个符号（ a 和 b ）进行计数，且两个符号均须计数到 2，由于除计数之外没有其他需求，因而所需的非终结符总数为 $2 \times 2 = 4$ 。如果规格说明又改为要求 a 的数目恰好被 3 整除，那么需要 3 个非终结符为 a 计数，并且这 3 个非终结符中的每一个都需要 2 个非终结符为 b 计数，故所需非终结符的总数为 6。

正则文法还可在生成的串中强制规定某些符号按指定的次序出现。如果一个非空串由任意数目的 a 、 b 和 c 组成，但 a 必须在所有 b 之前、 b 必须在所有 c 之前，则以下的简单文法即可

满足要求：

$A \rightarrow a A$	$A \rightarrow a$
$A \rightarrow b B$	$A \rightarrow b$
$A \rightarrow c C$	$A \rightarrow c$
$B \rightarrow b B$	$B \rightarrow b$
$B \rightarrow c C$	$B \rightarrow c$
$C \rightarrow c C$	$C \rightarrow c$

此处使用一个非终结符对 c 的数目进行计数（最多到 1），另一非终结符对第一个 c 之前的 b 计数，再一个非终结符生成第一个 b 之前的 a 。由于有 3 段字母（先是 a 的，再是 b 的，最后是 c 的），而每段计数只是 1，因而共需要 3 个非终结符，即三段计数之和。如果第一段要求有奇数个 a ，那么它将需要 2 个非终结符，所需非终结符的总数为 $2 + 1 + 1 = 4$ 。

从上述最后一个例子也可能体会到“文法思维”的另一个重要方面。由于非终结符 A 是目标符号，它表示语言中所有的串，可将该非终结符理解为它表示了“任意数目的 a 、后接任意数目的 b 、再接任意数目的 c ”；非终结符 B 表示仅由 b 和 c 组成的子串，其中不含 a ；一旦生成了第一个 b 就不会再有 a ，非终结符 B 记住了这一点。类似地，非终结符 C 记住了已生成一个 c ，因而不会再有任何 a 或 b 。

在再前的例子中，非终结符 A 表示由偶数个 a 和奇数个 b 组成的任意串（尽管尚未生成）； B 表示那些由奇数个 a 和奇数个 b 组成的未生成串，从而一旦 A 生成了单个 a ，剩余的未生成串中就少了一个 a ，因而它有奇数个 a ，这正是由 B 所表示的串。因而，如果一条产生式用一个已生成的 a 重写 A ，则其右部包含了非终结符 B 。

2.8.2 上下文无关文法的计数

由于下推自动机有一个无穷深度的栈，它可顺利地输入串中任意数目的符号进行计数，将它们与数目相同的另一些符号配对。这里所说的“计数”并不是指将符号的数目以某种数值表示存储起来，数学意义上的计数是指基数的一一对应，因而我们的重点在于匹配方面。例如，若要不一个文法对 a 和 b 计数，从而强制规定它们有相同的数目，则只需一个简单的上下文无关文法即可完成：

$A \rightarrow a A b A$
$A \rightarrow b A a A$
$A \rightarrow a b$
$A \rightarrow b a$

尽管该文法是二义的，但它仍演示了计数的一个重要特性：由于在有选择时，我们无法强制规定选择哪一条产生式（除非正试图匹配一个特定的终结字符串），因而同一产生式中两个符号出现的次数必须相同，才可保证所有的符号匹配。上述例子中，由于无法强制规定选择四条产生式中的哪一条（因为这四条产生式均重写了非终结符 A ），获得相同数目 a 和 b 的惟一方法，就是对每一条出现了 a 或 b 之一的产生式，在这同一条产生式中各放一个 a 和一个 b 。照此方法，就不会生成一个 a 而不同时生成一个 b 。

在实际应用中，惟一必须做到的就是待配对的终结符（好像）是从同一“共同祖先”推导出的；所谓“共同祖先”即单条产生式，它恰好推导出每一个符号。例如，之前介绍的上下文

敏感文法 G_3 生成相同数目的 a 、 b 和 c ，但在同一产生式中同时出现的三个符号由终结符 a 以及非终结符 B 和 C 组成，这两个非终结符恰好分别生成一个终结符（各自对应的小写字母），因而它们可有效地匹配。

文法 G_2 生成加法和乘法的所有算术表达式，其中采用习惯上的运算符优先级，但可使用圆括号重新规定优先级。再次提醒为非终结符赋予尽量有含义的名字的重要性，例如 E （表达式 Expression 的首字母）、 T （项 Term 的首字母）和 F （因子 Factor 的首字母），从而可反映这些非终结符表示了所生成的语言中有意义的组成部分。

E	E
$E + T$	T
$T + T$	F
$F + T$	(E)
$F + F$	(T)
$(E) + F$	(F)
$(E) + (E)$	$((E))$
$(T) + (E)$	$((E + T))$
$(T) + (T)$	$((T + T))$
$(F) + (T)$	$((F + T))$
$(F) + (F)$	$((F + F))$
$(n) + (F)$	$((n + F))$
$(n) + (n)$	$((n + n))$

图 2-10 生成正确嵌套括号的两个推导过程

让我们先探讨圆括号的配对问题。留意 G_2 只会生成那些成对匹配的圆括号，因为左、右括号在同一产生式中各自仅出现一次（规则 D.5）。此外，左括号总是在与它配对的右括号之前，它们在产生式中的次序强制规定了这一点。这并不意味着所有左括号都必须出现在所有右括号之前，实际情况也并非如此：像 $((n + n))$ 一样，串 $(n) + (n)$ 也属于该语言。然而，追踪这两个串的生成过程（如图 2-10 所示）可看出，生成这两个串时括号总会正确地嵌套在一起。

这种“正确的嵌套”是上下文无关文法的特征之一：我们可要求符号按正确的嵌套次序配对，但无法要求它们按相同的从左到右次序配对。我们可写出一个上下文无关文法生成回文数（串的前一半与后一半的逆序配对），但无法写出一个上下文无关文法匹配同一单词拼写的两次出现，要做到这一点需要一个上下文敏感文法。

例如，假设要写出一个上下文无关文法生成如此组成的串：任意数目、任意次序的 a 和 b ，后接单个 c ，最后再接与第一段相同的 a 和 b 序列（次序也相同）。记住为得到与第一段串匹配的第二段串，我们必须在同一产生式中生成每段各一个的配对符号，我们的尝试可能有些类似文法 G_{12} ：

$$A \rightarrow a A a$$

$$A \rightarrow b A b$$

$$A \rightarrow c$$

很不幸，这个文法生成的是第二段作为第一段倒置的回文数。如果试图修改前两条产生式，使得非终结符 A 位于左端或右端，那么两段符号串就无法按要求以中间的 c 分隔。添加另外的非终结符也不能解决这一问题。

上述分析中有一点是显而易见的：递归的产生式（生成任意长度的串的惟一途径）将生成递归的非终结符之外的所有其他符号的多个副本；产生式中这些符号出现在非终结符的哪一侧，就从那一侧弹出其副本。因而递归的产生式：

$$A \rightarrow a A b c$$

沿左侧生成一个个 a 、沿右侧生成一对对 bc ，如图 2-11 所示。由于每次应用这一递归的产生式会在待处理串中产生非终结符的另一副本（在本例中为 A ），因而必须至少有另一个非递归的产生式重写这个非终结符，例如 $A \rightarrow e$ ，否则这一递归的产生式不会生成任何串。非递归的产生式生成的任何东西都将出现在所生成串的左段与右段的中间。在文法 G_{12} 的回文数例子中，惟一非递归的产生式恰好生成一个 c ，这正是回文数的中点。这是一个相当重要的概念，有助于我们稍后学习正则表达式以及构建 $LL(k)$ 分析程序。

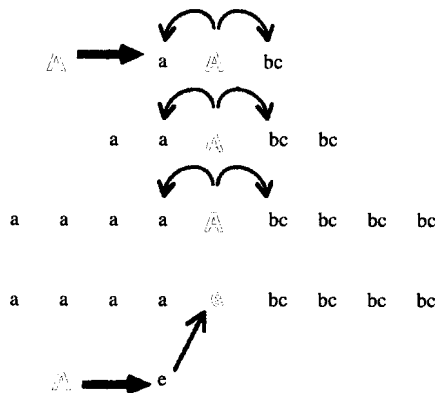


图 2-11 生成带终结符的配对重复模式

2.8.3 对上下文敏感

尽管我们并不打算从上下文敏感文法自动地构造线性有界自动机（LBA），但了解此类文法提供了上下文无关文法无法提供的哪些东西，有时会带来一些启示。如前所述，需要上下文敏感性的特性之一是以相同次序重复出现符号串的能力。上下文敏感文法 G_{13} 生成的语言是由 a 和 b 的相同子串组成的所有对偶，每一子串均以单个 c 开头并结尾；文法如下所示：

$$G_{13} = (\{a, b, c\}, \{S, A, B, F, G\}, P, S)$$

其中， P 是图 2-12a 所示的产生式集合。

$S \rightarrow c G F$	\underline{S}
$G \rightarrow a G A$	$c \underline{G} F$
$G \rightarrow b G B$	$c a G \underline{A} F$
$G \rightarrow c c$	$c a \underline{G} a F$
$F \rightarrow c$	$c a b G \underline{B} a F$
$A a \rightarrow a A$	$c a b G a \underline{B} F$
$A b \rightarrow b A$	$c a b G a b F$
$B b \rightarrow b B$	$c a b b G \underline{B} a b F$
$B a \rightarrow a B$	$c a b b G a \underline{B} b F$
$A F \rightarrow a F$	$c a b b G a b \underline{B} F$
$B F \rightarrow b F$	$c a b b G a b b F$
	$c a b b c c a b b \underline{F}$
	$c a b b c c a b b c$
a)	b)

图 2-12 文法 G_{13} 的产生式，见图 a；以及串 $cabbccabbc$ 的推导过程

非终结符 G 生成成对的符号 (a, A) 或 (b, B) ；非终结符 F 标记最右端子串的结尾，也作为上下文告诉非终结符 A 和 B 何时转为（推导出）终结符。 A 和 B 在串的中部生成后，在向右“漫步”时与所有终结符 a 和 b 交换位置，直至到达标记 F 。最后当串完成时， G 和 F 转为

c. 如果 F 太快转为终结符, 则剩余的 A 或 B 无法生成终结符 (即推导过程不会生成任何句子); 类似地, 直至 A 和 B 最终到达串的右端标记之前, 它们不可转为终结符。尽管在先前生成的 A 和 B 到达结尾并被转换之前, 可从中间生成更多的 A 和 B, 但它们无法打乱 A 和 B 的原有次序, 因为只有在较早生成的非终结符 A 或 B 到达其最终驻留的位置并成为终结符之后, 后生成的 A 和 B 才可能进行交换并转为终结符。

这是一个有趣的问题, 主要因为它是语言中“先声明、后使用”需求的一种极其简化的表示。令 c 代表程序的所有其他部分, 第一个由 a 和 b 组成的子串是某一标识符命名的变量或过程声明, 第二个子串是程序体中对它们的使用。要让文法形式化地校验每一标识符在使用前均有声明, 就必须像这样在句子中以相同方式让一个子串出现两次。过程调用中的参数类型检查本质上也是同一问题; 由于上下文无关文法无法做到这一点, 我们可断言上下文无关文法无法检查变量和过程声明的类型。在研究人员找到一种有效途径将上下文敏感文法转换为确定的自动机, 以及积累了编写上下文敏感文法定义这些语言需求的大量经验之前, 我们必须找出另外的方法处理变量声明的检查问题。所有现代编译程序都为此使用了一个带符号表的约束程序。

这一特殊例子还演示了开发一个切实可行的上下文敏感文法的一些典型编写风格。关键的思路是在非终结符这一生成器还活跃的地方 (典型情况是像上下文无关文法中用到的递归) 生成符号 (或代表这些符号的非终结符), 然后使用上下文敏感的产生式将仍处于非终结符形态的符号排序或重组。只需包含某些可能的符号排列变换, 即可恰好取得所需的效果。最后, 在刚开始时生成的某一标记可触发向终结符的转换; 如有必要, 这种转换可像推倒多米诺骨牌一样传播。图 2-13 展示了文法 G_3 的传播路径, 这是此类文法思维的另一实例。

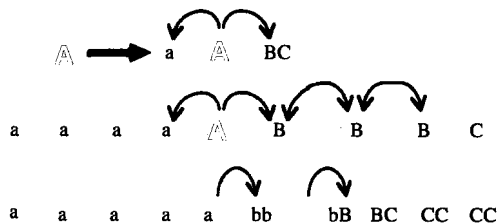


图 2-13 在上下文敏感文法中对符号排序

小结

文法是语言的数学描述, 这一描述的形式是一个四元组 (Σ, N, P, S) , 其中 Σ 是字母表, N 是非终结符的集合, P 是产生式的集合, S 是一个目标符号。编写文法是一种艺术, 这意味着它不仅依赖于语言原理和设计技巧方面的知识, 还取决于一些只可意会、不可言传的创造性, 而创造性更多是源于实践而不是教学。编写文法的设计技巧源自对语言结构的感觉和对语言形态的洞察, 使得设计人员能够将字母表和非终结符塑造为一门语言的描述。

乔姆斯基层次有效地组织了我们要处理的语言, 本章将乔姆斯基层次中的语言与语言识别器及其对应的文法联系在一起。本书的重点是正则文法和上下文无关文法的机械化实现, 但本章还使用了上下文敏感文法帮助理解其中的艺术性。上下文敏感文法仅对其产生式右部的长度有限制, 即对每一产生式 $\alpha \rightarrow \beta$, β 中符号 (终结符与非终结符) 的数目必须至少要有 α 中那么多, 惟一允许的例外是语言中的空串。上下文无关语言是上下文敏感语言集合的一个真子集。上下文无关语言的文法比上下文敏感语言的文法有更多的限制, 即上下文无关文法的每一产生式的左部必须恰好只有一个非终结符 (不可有终结符)。正则语言的文法有更多的限制, 在箭头右边最多只能有一个非终结符, 并且非终结符必须全部在最右端。

语言识别器是定义一个串的集合的过程。识别器的建模包括三个基本部分: 输入带, 有穷状态的控制单元, 并可能有某种形式的存储。控制单元类似于一个计算机程序, 它根据读入的当前输入符号, 决定输入头在带上移动的方向。上下文无关语言的识别器是下推自动机 (PDA), 得名于其存储单元——栈。

正则语言的识别器是有穷状态自动机 (FSA), 这是一种最简单的识别器。有穷状态自动机没有存储

单元。给出一个有穷状态自动机的规格说明时，要定义一个控制状态的有穷集、一个输入符号集、一个起始状态，以及一个有穷的终结状态集。终结状态指示接受一个输入串。有穷状态自动机为定义一门正则语言中串的集合提供了另一条途径。

总而言之，本章介绍了两种方式定义语言中的串：识别器和文法。在实际应用中，编译程序是一个识别器，可根据对应的文法机械地构造出来。因而，砺练我们的文法编写技巧远比直接关注自动机的设计技术更为重要。

符号

Σ 可用于构造某一特定语言中句子的所有输入字符或符号的有穷集。

Σ^* 由 Σ 中 0 个或多个符号组成的所有可能串的集合。

ϵ 空串。

N 非终结符（不属于 Σ ）的集合。非终结符是元语言符号，表示 Σ^* 的子集。

P 文法的重写规则（产生式）集合。

\Rightarrow 推导，推导过程中的单个步骤。

\Rightarrow^* 0 步或多步推导。

缩略词

ASCII American Standard Code for Information Interchange, 美国信息交换标准编码，是计算机中字母和数字等文本字符的一种常见表示。

CFG Context-Free Grammar, 上下文无关文法。

FSA Finite-State Automaton, 有穷状态自动机，正则语言的识别器。

FSM Finite-State Machine, 有穷状态机，等价于 FSA。

LBA Linear-Bounded Automaton, 线性有界自动机，上下文敏感语言的识别器。

PDA Push-Down Automaton, 下推自动机，上下文无关语言的识别器。

TM Turing Machine, 图灵机，在所有自动机中受到的限制最少。

关键术语

alphabet（字母表） 即符号集 Σ ，语言中的句子用其中的符号构成。

ambiguous grammar（二义文法） 指文法的语言中至少有一个句子存在多于一种的分析，或该句子至少有两种最左或最右推导。

derivation（推导） 是推导语言中的一个串的一系列步骤，每步应用一条重写规则。

canonical（规范推导） 以指定次序应用规则的两种推导之一。

leftmost（最左推导） 推导中每一步都替换最左的非终结符。

rightmost（最右推导） 推导中每一步都替换最右的非终结符。

frontier（边缘） 分析树的叶结点，表示一个句子。

grammar（文法） 是一个四元组 (Σ, N, P, S) ，其中，

Σ = 字母表（终结符的集合）

N = 非终结符的集合（元语言符号）

P = 产生式（重写规则）的集合

S = 目标符号（一个非终结符）

grammar types（文法类别）：

context free（上下文无关文法） 这类文法允许在应用产生式时，不必考虑待被重写的非终结符之

外的其他上下文或符号。每一产生式形如 $A \rightarrow \beta$, 其中 $A \in N$, $\beta \in (N \cup \Sigma)^*$ 。

context sensitive (上下文敏感文法) 文法中的每一产生式形如 $A \rightarrow B$ 且 $|A| \leq |B|$, 其中 $|A|$ 表示 A 的长度。注意, 这意味着 B 不可为空串, 除非 A 是目标符号且不出现在任何产生式的右部。

left linear (左线性文法) 文法中的所有产生式形如 $A \rightarrow Bx$ 或 $A \rightarrow x$, 其中 $x \in \Sigma^*$, $A, B \in N$ 。

regular (正则文法) 是一种右线性文法, 其中所有的产生式形如 $A \rightarrow xB$ 或 $A \rightarrow x$, 其中 $x \in \Sigma$, $A, B \in N$ 。若 A 是目标符号且不出现在任何产生式的右部, 则允许产生式 $A \rightarrow \varepsilon$ 。

right linear (右线性文法) 文法中的所有产生式形如 $A \rightarrow xB$ 或 $A \rightarrow x$, 其中 $x \in \Sigma^*$, $A, B \in N$ 。

unrestricted (无限制文法) 文法中的产生式形如 $\alpha \rightarrow \beta$, 其中 α, β 是文法符号的任意串, 且 $\alpha \neq \varepsilon$ 。

language (语言)

- (1) 字母表 Σ 上有穷长度的串的集合。
- (2) 一个 Σ^* 的指定子集。
- (3) 一个文法生成的串的集合 (参阅 **grammar** 条目)。
- (4) 一个语言识别器接受的输入串的集合 (参阅 **recognizer** 条目)。

language type (语言类别)

- 0 (0 型语言) 由一个无限制文法生成的短语结构语言。
- 1 (1 型语言) 由一个上下文敏感文法生成的上下文敏感语言。
- 2 (2 型语言) 由一个上下文无关文法生成的上下文无关语言。
- 3 (3 型语言) 由一个正则文法生成的正则语言。

nonterminal (非终结符) 一个不属于 Σ 的标识符或符号, 但代表了 Σ^* 的一个子集。

parse tree (分析树) 用一个二维形式的图, 表示将目标符号重写为一个句子的所有步骤。也称抽象语法树 (AST)。

production (产生式)

- (1) 一条重写规则, 描述语言中的句子如何生成的一个步骤。
- (2) 一对串 $\alpha \rightarrow \beta$, 其中 α 包含至少一个非终结符, β 可以是终结符和非终结符的混合串。

recognizer (识别器) 一个恰好接受 (识别) 语言中所有串的真实机器或抽象机器。

recognizer type (识别器类别):

- 0 (0 型识别器) 即图灵机 (TM), 定义了一种短语结构语言或递归可枚举语言。
- 1 (1 型识别器) 即线性有界自动机 (LBA), 定义了一种上下文敏感语言。
- 2 (2 型识别器) 即下推自动机 (PDA), 定义了一种上下文无关语言。
- 3 (3 型识别器) 即有穷状态自动机 (FSA), 定义了一种正则语言。

sentence (句子) 语言中的一个串。

sentential form (句型) 推导过程中的任意一行。

stack (栈) 一种线性数据结构, 其中所有插入 (压入) 和删除 (弹出) 操作都作用在该数据结构的一端。

terminal (终结符) 一个用于构造语言中的句子的符号, 是字母表 Σ 中的一个元素。

vocabulary (词汇表) 终结符和非终结符的集合 (Σ 和 N)。

练习

1. 指出以下陈述是否正确:

- (a) 所有上下文无关文法都是上下文敏感的。
- (b) 所有右线性文法都是上下文无关的。
- (c) 所有正则文法都是右线性的。
- (d) 所有正则文法都是左线性的。

(e) 所有左线性文法都是上下文无关的。

(f) 所有左线性文法都是上下文敏感的。

(g) 所有正则文法都是上下文敏感的。

(h) 所有右线性文法都是上下文敏感的。

2. 将下列文法划分为无限制文法、上下文敏感文法、上下文无关文法、右线性文法、左线性文法或正则文法。一些文法可归入多类，请指出每一文法所属的所有类别：

(a) $S \rightarrow aB$

$S \rightarrow c$

$C \rightarrow cA$

$B \rightarrow bC$

$B \rightarrow b$

$A \rightarrow aA$

$A \rightarrow a$

(e) $S \rightarrow ab$

$S \rightarrow B$

$B \rightarrow bc$

$B \rightarrow \epsilon$

(i) $S \rightarrow ab$

$S \rightarrow B$

$B \rightarrow bc$

$B \rightarrow ab$

(b) $S \rightarrow Cba$

$S \rightarrow C$

$C \rightarrow Bc$

$B \rightarrow Cb$

$B \rightarrow b$

$A \rightarrow Ba$

$A \rightarrow Aa$

(f) $S \rightarrow aA$

$S \rightarrow \epsilon$

$A \rightarrow bcs$

(j) $S \rightarrow Aab$

$S \rightarrow B$

$A \rightarrow Ba$

$B \rightarrow bc$

$B \rightarrow \epsilon$

(c) $S \rightarrow abc$

$S \rightarrow A$

$A \rightarrow aB$

$A \rightarrow aA$

$B \rightarrow bC$

$B \rightarrow b$

$C \rightarrow c$

(g) $S \rightarrow a$

$S \rightarrow b$

$S \rightarrow \epsilon$

(k) $S \rightarrow ABC$

$AB \rightarrow A$

$A \rightarrow aB$

$aBC \rightarrow abC$

$C \rightarrow cd$

(d) $S \rightarrow ABC$

$S \rightarrow A$

$A \rightarrow aBA$

$A \rightarrow a$

$B \rightarrow BC$

$B \rightarrow b$

$C \rightarrow c$

(h) $S \rightarrow abc$

$S \rightarrow B$

$B \rightarrow bc$

(l) $S \rightarrow aAbc$

$S \rightarrow \epsilon$

$A \rightarrow Bc$

$B \rightarrow bB$

$B \rightarrow cS$

3. (a) 指出以上文法中哪些是二义的（如果有的话）。

(b) 证明你在 (a) 中所选的文法是二义的。

4. (a) 为文法 G_2 构造推导出串 $(n+n)*n$ 的分析树，并分别给出最左推导和最右推导。

(b) 说明文法 G_2 无法推导出串 $n+*n$ 。

(c) 说明文法 G_3 无法推导出串 $aaccbb$ 和 $abbccc$ 。

5. (a) 构造一个正则文法，生成由 a 和 b 组成的所有串，每个串含有奇数个 a 和奇数个 b 。

(b) 用你构造的文法推导出串 $aababb$ 。

6. (a) 构造一个正则文法，生成由 a 和 b 组成的所有串，其中所有 a 都按 3 个一组的方式出现。

(b) 用你构造的文法推导出串 $baaabaaabb$ 。

7. (a) 构造一个正则文法，生成 0 和 1 的所有可能组合，并且串的长度不超过 6 个字符。

(b) 用你构造的文法推导出串 01001。

8. (a) 为 Modula-2 语言的字符串常量编写一个正则文法。一个字符串常量的组成是由单引号括住的、且其中不再包含单引号的任意字符串，或由双引号括住的、且其中不再包含双引号的任意字符串。为简化问题，假设字母表中仅有字符 a 、 b 、' 和 "。

(b) 用你构造的文法推导出串 $'aba"aab''$ 。

9. (a) 编写一个上下文无关文法，生成由 a 和 b 组成的所有回文。回文是一个正向读和反向读都一样的串（例如 $abaabababaaba$ ）。

(b) 用你构造的文法推导出串 $abbaabba$ ，并画出其分析树。

10. (a) 为 Modula-2 语言的注释编写一个上下文无关文法。注释必须用 “(” 开头、用 “)” 结尾。为简化问题，假设仅有字符 a 、 b 、*、(和)。注意注释是可以嵌套的。

(b) 用你构造的文法推导出串 $(*ab*a**(b*))$ ，分别给出最左和最右规范推导，并画出其分析树。

11. (a) 编写一个上下文无关文法生成由 a 和 b 以任何次序组成的串, 使得每个串中 a 的数目多于 b , 这也意味着必须至少有一个 a 。
 (b) 用你构造的文法推导出串 $baaba$, 分别给出最左和最右规范推导, 并画出其分析树。
12. (a) 编写一个上下文敏感文法生成由 a 、 b 和 c 以任何次序组成的串, 使得每个串中 a 的数目多于 b 、 b 的数目多于 c , 这也意味着必须至少有两个 a 、至少有一个 b 。
 (b) 用你构造的文法推导出串 $caabaaba$ 。
13. (a) 编写一个上下文敏感文法生成由 a 、 b 和 c 以任何次序组成的串, 使得每个串中 a 、 b 和 c 的数目相同。
 (b) 用你构造的文法推导出串 $cacbab$ 。
14. (a) 编写一个上下文敏感文法生成由 a 和 b 组成的串, 使得每个串由两个相同的子串组成。例如, $aabaab$ 就是一个这样的串。
 (b) 用你构造的文法推导出串 $babbbabb$ 。
15. (a) 编写一个上下文敏感文法生成由 a 和 b 组成的串, 使得每个串由三个相同的子串组成。例如, $aabaabaab$ 就是一个这样的串。
 (b) 用你构造的文法推导出串 $abaaabaaabaa$ 。
16. (a) 为 Pascal 语言的实数常量 (例如 5 、 $4E5$ 、 $+3.8$ 、 $-29.6E15$ 、 $7E-3$) 编写一个右线性文法。注意: 小数点的前后总有数字; 幂部分最多只有两位数字。为简化问题, 令 d 代表任意数字。
 (b) 用你构造的文法推导出串 $-49.72E-12$, 分别给出最左和最右规范推导, 并画出其分析树。
17. 描述以下文法生成的语言。尽量用自然语言捕捉这些语言的本质, 而不是仅仅用自然语言改写这些文法。

- | | | | | |
|------------------------|------------------------|--------------------------|-----------------------|-----------------------|
| (a) $S \rightarrow cA$ | (b) $S \rightarrow 0S$ | (c) $S \rightarrow abcA$ | (d) $S \rightarrow 0$ | (e) $S \rightarrow a$ |
| $A \rightarrow cA$ | $S \rightarrow S0$ | $S \rightarrow Aabc$ | $S \rightarrow 1$ | $S \rightarrow *SS$ |
| $A \rightarrow 0A$ | $S \rightarrow 1S$ | $A \rightarrow \epsilon$ | $S \rightarrow 1S$ | $S \rightarrow +SS$ |
| $A \rightarrow 1A$ | $S \rightarrow S1$ | $Aa \rightarrow Sa$ | | |
| $A \rightarrow c$ | $S \rightarrow 0$ | $cA \rightarrow cS$ | | |
| $A \rightarrow 0$ | $S \rightarrow 1$ | | | |
| $A \rightarrow 1$ | | | | |

18. 为以下每种语言选择合适的文法级别和自动机。
- (a) 由 0 和 1 按任何次序组成的所有串, 每个串中 0 和 1 的数目相同。
- (b) 由 0、1 和 2 按任何次序组成的所有串, 每个串中 0、1 和 2 的数目相同。
- (c) 由 a 和 b 的两个相同子串组成的所有串, 每一子串中 a 和 b 可以任意次序出现 (例如 $abaaabaa$)。
- (d) 由一个 a 、一个 b 和两个 c 按任何次序组成的所有串 (例如 $cbca$)。
- (e) 由配对的圆括号组成的所有串 (例如 “ $((()))$ ”)。
- (f) 由成对的 0 和 1 组成的所有串, 即 0 紧贴在与它配对的 1 之前或之后。
19. 将以下上下文敏感文法改写为等价的上下文无关文法。

$S \rightarrow ABS$
 $S \rightarrow AB$
 $AB \rightarrow BA$
 $A \rightarrow 0$
 $B \rightarrow 1$

复习小测验

指出下列陈述是否正确:

1. 我们可定义一个有穷状态机识别空语言。
2. 仅由空串组成的语言是乔姆斯基层次中的 3 型语言。
3. 不可能构造一个有穷状态机识别以下文法定义的串：

$$A \rightarrow 0A \mid 1B \mid 0 \mid \varepsilon$$

$$B \rightarrow 1B \mid \varepsilon$$

4. 问题 3 中的文法是上下文无关的。
5. 问题 3 中的文法所定义的语言是正则的。
6. 问题 3 中的文法是左线性的。
7. 每一个右线性文法都是正则的。
8. 对一种正则语言而言，最多只有一个有穷状态机可输入并接受该语言中的串。
9. 每一个正则文法都是右线性的。
10. 每一个文法定义的语言都有一个对应的识别器。

编译程序实验项目

1. 编写一个正则文法生成如下 Itty Bitty Modula 语言的每一类单词。
 - (a) 所有标识符。
 - (b) 所有保留字。假设标识符 **INTEGER**、**BOOLEAN**、**TRUE** 和 **FALSE** 是保留字（在标准 Modula-2 中它们是预先声明的标识符）。
 - (c) 无符号整数，以及十六进制和八进制常量。
 - (d) 所有运算符和标点符号。
 - (e) 字符串常量（参见练习 8）。
 - (f) Itty Bitty Modula 语言的注释，但不允许子串 “(*)” 和 “(*)” 出现在注释中。
2. 编写一个上下文无关文法，生成 Itty Bitty Modula 语言的短语结构；必要时，可参阅附录 A 的语法图。可使用你在第 1 章结束时准备的单词列表作为你的字母表。
3. 用 Itty Bitty Modula 语言编写一个小程序，再用你的文法构造它的推导树，并为你的程序分别给出最左和最右规范推导。

进一步阅读

- Aho, A.V. & Ullman, J.D. *The Theory of Parsing, Translation, and Compiling: Vol. 1, Parsing*. Englewood Cliffs, NJ: Prentice Hall, 1972.
- 参阅第 2 章“语言理论基础”，特别是第 2.1 节关于识别器的介绍，以及第 2.2 节对正则语言的生成器和识别器的介绍。
- Chomsky, N. "Three Models for the Description of Language." *IRE Transactions on Information Theory*, Vol.2, No.3 (1956), pp.113-124.
- Chomsky, N. & Miller, G.A. "Finite State Languages." *Information and Control*, Vol.1, No.1 (1963), pp.91-112.
- Cohen, D.I.A. *Introduction to Computer Theory*. New York: Wiley, 1986.
- 参阅第 5 章关于有穷自动机的介绍，以及第二部分“下推自动机理论”，特别是第 19 章对上下文无关语言的介绍。也可参阅第 30 章“乔姆斯基层次”。
- Fass, L.F. "Learning Context-Free Languages from Their Structured Sentences." *ACM SIGACT News*, Vol.15, No.3 (1983), pp.24-35.
- 所谓“学习”问题主要涉及一些设计技术，这些技术可基于样本信息确定生成一个上下文无关语言 L 的文法 G。该文非常出色，因为它为本书第 2 章介绍的概念提供了一种形式化处理方法。
- Fass, L.F. "On the Inference of Canonical Context-Free Grammars." *ACM SIGACT News*, Vol.27, No.7 (Spring

1986), pp.55-60.

该文介绍了上下文无关文法的各种形式。

Haines, L. *Generation and Recognition of Formal Languages*. Ph.D. dissertation, MIT, 1965.

Hopcroft, J.E. & Ullman, J.D. *Introduction to Automata Theory, Languages, and Computation*. Reading, MA: Addison-Wesley, 1979.

参阅第9章“乔姆斯基层次”，特别是第9.1节“正则文法”和第9.2节“无限制文法”。

Parikh, R.J. "On Context-Free Languages." *Journal of the ACM*, Vol.13, No.4 (1966), pp.570-581.

Reis, A. "Regular Languages under F-gsm Mappings." *ACM SIGACT News*, Vol.18, No.3 (Spring 1987), pp.41-45.

F-gsm 是一种带终结状态的通用顺序机，其中与一个变迁相关联的输出可以有任意长度，并且仅当输入导致从起始状态转到终结状态时，才为该输入定义了输出。

Stearns, R.E. "A Regularity Test for Pushdown Machines." *Information and Control*, Vol.11 (1967), pp.323-340.

该文讨论了如何确定由一个下推自动机识别的输入串集合是否正则的，做法是提取一个与下推自动机等价的有穷状态机，它可识别正则语言中的串。

Tennent, R.D. *Principles of Programming Languages*. Englewood Cliffs, NJ: Prentice Hall, 1981.

参阅第1.3节关于短语结构的介绍，以及第2.6.2节关于二义语法描述的介绍。

Whitney, G.E. "The Generation and Recognition Properties of Table Languages." *Information Processing 68*. Amsterdam: North-Holland, 1969, pp.388-394.

第3节介绍了语言中串的识别与串的生成可理解为对偶。

第3章 扫描程序和正则语言

本章旨在：

- 详细说明扫描程序的理论基础
- 介绍正则表达式和正则语言的形式化特性
- 揭示正则表达式代数的实际意义
- 描述正则表达式与正则文法之间的转换技术
- 从正则表达式或正则文法开发一个有穷状态自动机
- 揭示如何将有穷状态自动机实现为一个扫描程序
- 考虑扫描程序使用的字符串表的有效实现方法

3.1 词法分析简介

编译程序的前端读入并分析源程序文本，它的大多数运行时间花费在扫描程序的词法分析，即从输入文件中读入字符，然后将它们约简为可管理的单词（字或特殊符号）。因而，编译程序设计人员有义务尽力提高扫描程序的效率。同时，我们还关注是否有一个清晰的形式化定义可产生正确的实现；效率并不能替代正确性。

编译程序中扫描程序的定义从一个文法入手。扫描程序文法定义的语言是一个外部（文本）字母表中的字符组成的所有串的集合，这形成了待编译语言中的单词。例如，任一标识符的串都是一个单词，因而它也是扫描程序语言中的一个句子。扫描程序识别一个这样的单词，然后“停机”并接受它。

这是一门简单的语言，用一个正则文法足以完整地定义该语言中的所有串。因而，一个有穷状态自动机（FSA）足以实现扫描程序。

本章介绍了从正则文法自动构造有穷状态自动机的方法，并展示了从 FSA 的形式化规格说明编写一个扫描程序的几种途径。本章还介绍了如何为文法添加必要的语义动作，以支持编译程序可利用识别出来的单词，并展示了这些动作在扫描程序的实现中是如何被处理的。

然而本章首先介绍的是表达正则语言的另一种表示法，这种表示法可更直观地捕捉我们对单词形态的理解。本章还将说明这种表示法与正则文法是等价的。

3.2 正则表达式

考虑整数文字量，数值 0 和 384 都是整数文字量的例子。我们可用自然语言给出整型常量单词的非严格定义：“至少有一个十进制数字，后接 0 个或多个另外的数字”。一旦意识到上述短语“0 个或多个”与克林星号表示法有关系，就可将整型常量的定义表示为“ dd^* ”，其中 d 表示一个数字。因而，一个常量是 1 个数字、后接 0 个或多个数字。

类似地，在 Pascal 和 Modula-2 语言中，一个标识符由一个字母（用 a 表示它）、后接 0 个或多个字母或数字组成。如果使用竖杠“ $|$ ”表示“或”，则可用“ $a | d$ ”这种写法表示“要么一个 a ，要么一个 d ，但不能两者都是”这个意思。这样就可将标识符简洁地表示为“ $a(a | d)^*$ ”。这种紧凑的表示法称为正则表达式。正则表达式是定义正则语言的一种强大且直观的表达法。

多数人对正则表达式的熟悉源自流行的 UNIX 操作系统。由于本书采用的表示法与其他表示法略有不同，并为了帮助读者进一步熟悉作为本书核心的文法表示法，本章在介绍正则表达式时，利用一个上下文无关文法描述其形式：

$$RE = (\{ " | , * , (,) " , (" , \sigma \} , \{ RegExpn , Term , Primary , Factor \} , P , RegExpn)$$

其中， σ （读作 sigma）表示正则表达式所定义语言的字母表中的任一符号，P 是如表 3-1 所示的产生式集合。

表 3-1 正则表达式文法的产生式

3.1	RegExpn	→ RegExpn " " Term	{ 选择 }
3.2	RegExpn	→ Term	
3.3	Term	→ Term Primary	{ 连接 }
3.4	Term	→ Primary	
3.5	Primary	→ Factor "*"	{ 迭代 }
3.6	Primary	→ Factor	
3.7	Factor	→ "(" RegExpn ")"	{ 分组 }
3.8	Factor	→ σ	{ 任意终结符 }

除了用 σ 所表示的字母表中的字符之外，正则表达式还有四个元符号：“|”、“*”、“)”和“(”。上述文法给出了正则表达式的语法，但并未给出其含义。它们的含义（如表中的注释所示）相当简单：竖杠表示选择，即两者选其一；这意味着可随意选择竖杠右边的 Term，而不是竖杠左边的 RegExpn。由于产生式是递归的，前两条产生式合在一起可生成一个或多个 Term 组成的序列，中间由竖杠分隔；在整个序列中，恰好有一个 Term 被选中。因而，“ $a|b$ ”表示“要么是 a，要么是 b，但不能两者都是”；正则表达式“ $a|b|c$ ”表示“要么是 a，要么是 b，要么是 c，但只能是其中之一”。

类似地，一个 Term 是 Primary 的序列，它们之间没有任何标点符号或元符号分隔，这一序列表示连接。正则表达式“ ab ”表示“a 的后面跟着 b”。

一个 Factor 既可以是字母表的一个符号，也可以是一个带圆括号的表达式。如果 Factor 右侧附有一个星号，我们称其为迭代；星号表示 Factor 重复出现 0 次或多次。因而， a^* 表示“任意数目的 a，或者没有任何 a”。注意，星号运算符的优先级最高，因而序列 $a|bc^*$ 中只有 c 是迭代的；选择的优先级最低，因而 bc^* 合在一起作为同一正则表达式中 a 的另一选项。

有时会使用另外两个或三个元符号，为表达不同类别的迭代提供方便。为引入这些元符号，我们为文法添加两条产生式：

3.9 Primary → Factor "+"
3.10 Primary → Factor "?"

加号迭代运算符“+”表示 Factor 重复出现 1 次或多次（不像星号那样出现 0 次或多次）；问号表示出现 0 次或 1 次。这些是次要的迭代运算符，因为它们完全可以用原有文法中的术语来定义：

a^+ 表示 aa^*
 $a^?$ 表示 $a|\epsilon$

根据定义，即可推出 $(a^+)^? = (a^?)^+ = a^*$ 。

3.2.1 正则表达式代数

选择和连接运算表现出数学上的“域”的某些特征，具有与加法和乘法运算相同的指导性代数法则。如果一个二元运算 \oplus 对任意 a 和 b 均有 $a \oplus b = b \oplus a$ ，则称该运算是交换的；如果运算 \oplus 对任意 a 、 b 和 c 均有 $(a \oplus b) \oplus c = a \oplus (b \oplus c)$ ，则称该运算是结合的；如果运算 \oplus 和 \bullet 对任意 a 、 b 和 c 均有 $a \bullet (b \oplus c) = a \bullet b \oplus a \bullet c$ ，则称运算 \bullet 对运算 \oplus 是分配的。连接运算对选择运算是分配的，但反之不然，即 $a(b|c) = ab|ac$ 但 $a|bc \neq (a|b)(a|c)$ 。连接运算和选择运算都是结合的，故有 $(ab)c = a(bc)$ 且 $(a|b)|c = a|(b|c)$ ；然而仅有选择运算是可交换的： $a|b = b|a$ 但 $ab \neq ba$ 。选择运算没有幺元，而连接运算的幺元是空串 ϵ ，因而 $a\epsilon = \epsilon a = a$ 。在我们熟悉的算术运算法则中，容易遗忘的是选择运算的吸收律：对任意 a 均有 $a|a = a$ 。表 3-2 总结了正则表达式的一些代数恒等式。

表 3-2 正则表达式的代数恒等式

设 r 、 s 和 t 为任意正则表达式，则：		
1.	$r s = s r$	(选择运算的交换律)
2.	$r (s t) = (r s) t$	(选择运算的结合律)
3.	$r r = r$	(选择运算的吸收律)
4.	$r(st) = (rs)t$	(连接运算的结合律)
5.	$r(st) = rs rt$	(左分配律)
6.	$(s t)r = sr tr$	(右分配律)
7.	$r\epsilon = \epsilon r = r$	(连接运算的幺元)
8.	$r^*r^* = r^*$	(闭包运算的吸收律)
9.	$r^* = \epsilon r rr ...$	(克林闭包运算)
10.	$(r^*)^* = r^*$	
11.	$rr^* = r^*r$	
12.	$(r^* s^*)^* = (r^*s^*)^*$	
13.	$(r^*s^*)^* = (r s)^*$	
14.	$(rs)^*r = r(sr)^*$	
15.	$(r s)^* = (r^*s)^*r^*$	

下面演示这些恒等式如何用于操纵正则表达式。考虑如下正则表达式：

$a(ba|ca)|(aca|aba)a$

该正则表达式可借助上述恒等式按如下方式化简：

$$\begin{aligned} &= (aba|aca)|(aca|aba)a && (5 \text{ 和 } 6) \\ &= aba|(aca|(aca|aba)) && (2) \\ &= aba|((aca|aca)|aba) && (2) \\ &= aba|(aca|aba) && (3) \\ &= aba|(aba|aca) && (1) \\ &= (aba|aba)|aca && (2) \\ &= aba|aca && (3) \\ &= a(b|c)a && (5 \text{ 和 } 6) \end{aligned}$$

化简后的正则表达式更简洁、更可读地定义了原正则表达式定义的串的集合。



3.2.2 正则表达式的形式化特性

一个正则表达式可非形式化定义为一种紧凑的表示法，用于定义同一字母表上的串的集合。例如，如果字母表是 $\{0, 1\}$ ， $(0|1)^*$ 这种写法表示了由0和1组成的所有串。下面形式化地定义了正则表达式：

定义 3.1 正则表达式是一个形式化表达式，即：

- a) 字母表 Σ 中的单个字符
- b) 空串 ϵ
- c) 空集 $\{\}$

或者，给定 Σ^* 中串的集合 R 和 S ，可通过有穷步应用下列操作得到一个正则表达式：

并运算： $R|S = \{x | x \in R \text{ 或 } x \in S\}$

连接运算： $RS = \{xy | x \in R \text{ 且 } y \in S\}$

闭包运算： $R^* = \{\} | R | RR | RRR | \dots$

因而，实际上一个正则表达式可以是某一字母表中的单个字符，也可以通过组合一次或多次并、连接、闭包等运算建立起来。例如，字母表 $\{0, 1\}$ 中的单个字符“0”是一个正则表达式。注意，正则表达式“0”表示集合 $\{0\}$ ，它本身是一种语言；集合 $\{0\}$ 是正则语言的一个例子。

定义 3.2 正则语言是一个字母表上的串的集合 L ，并且 L 可以由一个正则表达式定义。

换言之，如果存在一个正则表达式可表达语言 L 中的串，那么语言 L 是正则的。这意味着如果 R 和 S 是正则表达式，那么 R 和 S 分别定义了正则语言 $L(R)$ 和 $L(S)$ 。还应注意，用于构造正则表达式的集合是从 Σ^* 中抽取的串的集合，从 Σ^* 中抽取的每一集合都可能是有穷的。这些集合值得关注，因为其中的每一个都是正则语言。可利用数学归纳法和以下引理证明这一论断是正确的。

引理 3.1 包含单个串的集合有一个对应的正则表达式。

【证明】该引理的证明需要对串的长度采用数学归纳法，证明留作练习 20。 □

在证明定理 3.1 的归纳步中需要用到该引理。

定理 3.1 每一个由串组成的有穷集都是正则语言。

【证明】(采用归纳法)

基本步：由定义，空集 $\{\}$ 是一个正则表达式；再由定义，一个正则表达式定义的串的集合是正则的，故 $\{\}$ 是一个正则语言。类似地，空串 ϵ 是一个正则表达式，因而集合 $\{\epsilon\}$ 也是正则的。最后，一个字符 a 是表示集合 $\{a\}$ 的正则表达式，所以 $\{a\}$ 是一个正则语言，因为它可以用一个正则表达式表示。

归纳假设：假设由 k 个元素的串组成的集合 L 可由正则表达式 r 表示，即假设 L 是一个正则语言。

归纳步：证明任意 $k+1$ 个元素的串的语言 L' 可由一个正则表达式表示。为证明这一点，可将 L' 表达为 L 和 $\{a\}$ 的并集，即将 L' 表达为长度为 k 的串的语言 L 和其中有一个串的语言 L'' （例如 $\{a\}$ ）的并集。由归纳假设可知， L 对应着正则表达式 r ；据引理 3.1 也可知，语言 L'' 有一个对应的正则表达式（称之为 r'' ）。据定义，可知 $r|r''$ 是一个正则表达式，这意味着语言 L' 有一个对应的正则表达式，这正是我们想证明的。 □

也可能有读者会问：定理 3.1 反过来是否也成立？命题 3.1 陈述了这一想法（命题是一个可能为真、也可能为假的断言）：

命题 3.1 每一正则语言都是有穷的。

【分析】参阅练习 21。□

最后，注意正则语言关于并运算、连接运算和克林星号运算都是封闭的，即可证明如果 L_1 和 L_2 均为正则语言，则可通过计算 $L_1 \cup L_2$ 或 $L_1 \cdot L_2$ 或 L_1^* 或 L_2^* 得到另一正则语言。这一论据总结如下：

定理 3.2 如果 L_1 和 L_2 是正则语言，则 $L_1 \cup L_2$ 、 $L_1 \cdot L_2$ 和 L_1^* 分别也是正则语言。

【证明】借助于正则语言的定义可证明该定理，证明的细节留作练习 22。□

集合 R 上的克林星号运算表示如下连接运算的无穷并集：

$$R^* = \{ \} \cup R \cup RR \cup RRR \cup \dots$$

也可定义稍小一些的并集，例如：

$$R^3 \cup R^5 = RRR \cup RRRRR$$

根据定义，我们有：

定义 3.3 给定语言 L ，令 $L^0 = \{ \epsilon \}$ ，即仅有一个空串的语言。

这产生了另一定理：

引理 3.2 如果 L 是一个正则语言，则 L^n 也是一个正则语言。

【证明】参阅练习 23。□

现在也可证明：

定理 3.3 如果 L 是一个正则语言，则下列语言也是正则语言：

(a) $L^n \cup L^m$ ，其中 $n, m \geq 0$

(b) $L^+ = \{ \epsilon \} \cup L$

3.3 文法与正则表达式的转换

正则语言既可由一个正则表达式定义，也可由一个正则文法定义。换言之，对每一正则文法，存在一个正则表达式定义相同的语言；对每一正则表达式，存在一个正则文法产生相同的语言。然而，某些正则语言用文法来定义会更容易，而另一些正则语言用正则表达式来定义会更清晰，因而两者之间能够相互转换是很重要的。下面通过构造性方法建立它们的等价性。

正则文法的产生式与正则表达式在一个关键方面有所区别：文法是一个由重写目标符号的规则组成的集合，而表达式仅仅描述所生成的串。正则表达式在描述方面显得更直观，即通过分析一个正则表达式往往比分析一个文法更容易看出语言中恰好有什么样的串。

将正则表达式转换为文法的第一步是将它改写为一条重写规则，添加一个目标符号即可做到这点。因而对任意正则表达式 ω ，选择某一非终结符 S 并令其为新文法中的目标符号，然后写出产生式：

$$S \rightarrow \omega$$

正则表达式通常含有一些或全部的元符号，而正则文法中未定义这些元符号，因而转换过程的第二步是消除这些元符号。

令 x 和 y 是任意正则表达式，可能是空表达式或包含非终结符。对每一个如下形式的产生式：

$$A \rightarrow xy$$

选择某一新的非终结符 B, 并改写为:

$$A \rightarrow xB$$

$$B \rightarrow y$$

对转换过程中得到的每一个如下形式的产生式:

$$A \rightarrow x^*y$$

改写为四条产生式:

$$A \rightarrow xB$$

$$A \rightarrow y$$

$$B \rightarrow xB$$

$$B \rightarrow y$$

仅当迭代运算嵌套在另一结构之中时, 才需要另外的非终结符。但多个非终结符只是冗长而已, 并不会带来其他害处。对以下形式的每一产生式:

$$A \rightarrow x|y$$

改写为:

$$A \rightarrow x$$

$$A \rightarrow y$$

连续执行上述转换, 如有必要可随时应用代数恒等式, 直至得到的文法是右线性的, 即其中不含正则表达式的元符号, 且每一产生式中最多只有一个非终结符。注意, 该构造过程保证了任一产生式的右部不会有多于一个的非终结符, 且非终结符将出现在最右端; 类似地, 任一产生式的右部最多只有一个终结符 (如果不是这样, 只需连续应用第一条转换规则, 直至结果如此)。表 3-3 将正则表达式转换为右线性文法的规则总结为规则 R.1~R.3。

表 3-3 将正则表达式转换为正则文法

规则#	正则表达式的产生式	文法的产生式	
R.1	$A \rightarrow xy$	$A \rightarrow xB$	$B \rightarrow y$
R.2	$A \rightarrow x^*y$	$A \rightarrow xB y$	$B \rightarrow xB y$
R.3	$A \rightarrow x y$	$A \rightarrow x$	$A \rightarrow y$
R.4	$A \rightarrow B$	$A \rightarrow x$	$B \rightarrow x$
R.5	$A \rightarrow \epsilon$	$B \rightarrow xA$	$B \rightarrow x$
R.6	$S \rightarrow \epsilon$ (S 是目标符号)	$G \rightarrow S$	$G \rightarrow \epsilon$

可能还存在一些产生式没有终结符, 导致文法虽然是右线性的, 但不是正则的。下面说明如何将一个右线性文法改写为正则文法。如果一个右线性文法的任何单条产生式中有多于一个的终结符, 可采用上述第一条规则对这些产生式进行转换。下面考虑不含终结符的产生式。对每一个如下形式的产生式:

$$A \rightarrow B$$

找出并复制所有 B 出现在左部的产生式, 在副本中用 A 替换 B, 然后删除这一有问题的产生式。

对于空产生式可采用第 2 章介绍的转换方法, 即对任一空产生式:

$$A \rightarrow \varepsilon$$

找出并复制所有 A 出现在右部的产生式，从副本中删除 A 。然后，若 A 不是目标符号，则删除这一空产生式；如果 A 是目标符号且 A 还出现在某一产生式的右部，选择一个新的非终结符 G 作为目标符号，并添加两条产生式：

$$G \rightarrow A$$

$$G \rightarrow \varepsilon$$

连续应用这两个转换规则，直至得到的文法是正则的。表 3-3 将右线性文法转换为正则文法的规则总结为规则 R.4~R.6。

例如，考虑将如下正则表达式转换为正则文法：

$$a(a|d)^*$$

开始时，加上一个目标符号 S ：

$$S \rightarrow a(a|d)^*$$

外层正则表达式结构是连接，因而应用规则 R.1：

$$S \rightarrow aA$$

$$A \rightarrow (a|d)^*$$

再对第二条产生式应用规则 R.2，其中 x 对应圆括号中的内容，而 y 为空，从而得到：

$$S \rightarrow aA$$

$$A \rightarrow (a|d)B \quad B \rightarrow (a|d)B$$

$$A \rightarrow \varepsilon \quad B \rightarrow \varepsilon$$

对两个选择运算应用分配律，接着应用规则 R.3：

$$S \rightarrow aA$$

$$A \rightarrow aB \quad B \rightarrow aB$$

$$A \rightarrow dB \quad B \rightarrow dB$$

$$A \rightarrow \varepsilon \quad B \rightarrow \varepsilon$$

此时文法已是右线性的。可再用规则 R.5 消除两个空产生式，可得正则文法：

$$S \rightarrow aA \quad S \rightarrow a$$

$$A \rightarrow aB \quad A \rightarrow dB$$

$$A \rightarrow a \quad A \rightarrow d$$

$$B \rightarrow aB \quad B \rightarrow dB$$

$$B \rightarrow a \quad B \rightarrow d$$

要将一个正则文法转换为正则表达式，只需将上述过程反过来。换言之，如果单个非终结符 A 有多于一个的产生式，则用单条产生式替换所有这些产生式，将所有右部合并为单个右部，中间用竖杠“|”分隔，如表 3-4 中规则 R.3 所示。如果得到的产生式是递归的，可应用交换律重组各个项，从而使得所有递归的非终结符放在一起，然后应用分配律将它们化为因子；再将剩下的项用结合律组织在一起，从而结果形如规则 R.2，然后用迭代的形式取而代之。这样就在整个子表达式的外面加上了单个星号，这一子表达式的前身为递归的产生式。除目标符号之外，任何仅有一条产生式的非递归的非终结符若出现在另一产生式的右部，则可根据规则 R.1 以用其右部替换该非终结符的所有出现。当目标符号定义了惟一的一条产生式，且其右部

已无任何非终结符时，其右部即是所需的正则表达式。

表 3-4 将正则文法转换为正则表达式

规则#	文法的产生式	正则表达式的产生式
R.1	$A \rightarrow xB \quad B \rightarrow y$	$A \rightarrow xy$
R.2	$A \rightarrow xA y$	$A \rightarrow x^*y$
R.3	$A \rightarrow x \quad A \rightarrow y$	$A \rightarrow x y$

考虑如下的简单正则文法，我们希望将它转换为正则表达式：

$$S \rightarrow aA \quad S \rightarrow a$$

$$A \rightarrow aA \quad A \rightarrow a$$

$$A \rightarrow dA \quad A \rightarrow d$$

首先对两个非终结符应用规则 R.3，从而各得一个“产生式”：

$$S \rightarrow aA|a \quad A \rightarrow aA|a|dA|d$$

根据交换律重组各个项，根据结合律添加圆括号，将 A 中所有递归的项一起放在左边：

$$S \rightarrow aA|a \quad A \rightarrow (aA|dA)|(a|d)$$

分配律可用于将递归的 A 化为因子，从而此时产生式的形式已可应用规则 R.2：

$$A \rightarrow (a|d)A|(a|d)$$

应用规则 R.2 可消除迭代符号的递归：

$$A \rightarrow (a|d)^*(a|d)$$

注意，上述转换同时消除了递归的非终结符 A 和其后的“或”竖杠。此时该正则表达式可替换 S 的产生式，得：

$$S \rightarrow a(a|d)^*(a|d)|a$$

此时得到的正则表达式有些冗长，但应用代数法则可在一定程度上将它化简。首先，对选择运算右边的 a 应用连接运算的幺元 $a = a\epsilon$ ，然后应用分配律将选择运算两边中开头的 a 化为因子，得：

$$a((a|d)^*(a|d)|\epsilon)$$

据 x^+ 的定义，可得：

$$a((a|d)^+|\epsilon)$$

然后立即可得：

$$a(a|d)^+$$

3.4 有穷状态自动机

对正则文法和正则表达式定义的每一个语言，存在一个确定的有穷状态自动机识别同一语言。有穷状态自动机定义为一个五元组，即它有五个不同的组成部分：

$$M = (\Sigma, Q, \Delta, q_0, F)$$

有穷状态自动机的字母表 Σ 与正则文法的相同（也与正则表达式规格说明中的所有 σ 的集合相同）；Q 是状态的有穷集，其中 q_0 是一个特殊的状态，称为起始状态；F 是 Q 的子集，称为终结状态集或停机状态集。停机状态是 FSA 可停机的任何状态。变迁规则的有穷集 Δ 定义了自动机如何基于输入带上的符号从一个状态前进到下一状态。变迁是关于当前状态和下一输入符

号的偏函数:

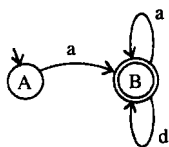
$$\Delta: Q \times \Sigma \rightarrow Q$$

对任意状态 A 和输入符号 a , 读入 a 后前进到状态 B 的变迁记为:

$$\delta(A, a) = B$$

格局是一对 (q, ω) , 其中 $q \in Q$ 是 FSA 的当前状态, $\omega \in \Sigma^*$ 是剩余的 (未读入的) 输入。当自动机处于格局 (q, ε) , 其中 $q \in F$ 是一个终结状态时, 自动机停机并接受输入串。换言之, 如果当输入串中不再有符号之时 FSA 处于一个停机状态, 则称该 FSA 接受该输入串; 否则, 如果 FSA 所处的状态没有针对下一输入符号定义了变迁, 则 FSA 阻塞并拒绝将这一输入串作为其语言中的一个成员。如果不再有输入符号, 但 FSA 不是在一个停机状态中, 它也会拒绝该输入串。

如果两个自动机接受相同的语言, 则称它们是等价的。如果它们等价, 且具有相同的状态和变迁, 仅仅是状态的名字不同, 则称它们是同构的; 换言之, 同构的 FSA 仅在其状态的某些名字上有所不同。如果一个 FSA 不存在比它更少状态的等价 FSA, 则称该 FSA 为约简的。图 3-1 中以图形方式展示的一个简单的有穷自动机是约简的, 图 3-2a 所示自动机与之同构, 图 3-2b 所示自动机与之等价。



	a	d
A	B	
*B	B	B

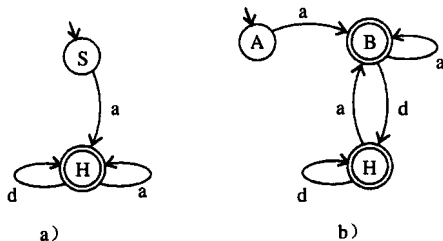


图 3-1 一个简单的有穷自动机, 分别以图形和表格表示

图 3-2 两个等价的有穷自动机

再次考虑由所有以 a 开头、并另外包含任意数目的 a 和 d 的串组成的语言。图 3-1 以图形方式展示了识别该语言的一个简单的双状态 FSA, 其形式化定义为:

$$M = (\{a, d\}, \{A, B\}, \{\delta(A, a) = B, \delta(B, a) = B, \delta(B, d) = B\}, A, \{B\})$$

字母表 Σ 由两个符号 a 和 d 组成; 有两个状态 A 和 B , A 是起始状态, B 是终结状态集中的惟一成员, 在图 3-1 中用双线圆表示; 有三条变迁, 每条变迁在图中用一条弧表示, 弧上标记了 FSA 执行变迁时依据的输入符号。该自动机是约简的, 因为不存在更少状态的等价自动机。

图 3-1 还以表格形式给出了同一 FSA, 表格中的行表示状态, 列表示输入符号。某一特定行与列的空白入口表示不存在变迁, 即此时 FSA 将阻塞。停机状态在其状态名字上用星号标记。

试对一个输入串 ada 运行该自动机。初始格局是 (A, ada) , 在状态 A 沿输入符号 a 有一个合法的变迁, 因而移动一步之后, 格局变为 (B, da) ; 下一变迁令其转入 (B, a) , 然后是 (B, ε) ; 由于 B 是一个终结状态, 因而该自动机接受输入串 ada 。如果打算对输入串 dd 运行同一 FSA, 会发现在状态 A 沿输入符号 d 不存在变迁, 因而该自动机阻塞并拒绝 dd , 认为它不属于该语言。

3.5 不确定的有穷状态自动机

尽管编译程序设计中仅对实现确定的自动机感兴趣, 但学习不确定的 FSA 仍是有益的, 这主要是因为从文法到扫描程序的转换具有不确定性。

在任意给定的状态、对任意给定的输入符号，确定的自动机最多只有一个可能的变迁；而不确定的有穷状态自动机（NFA）与确定模型在两方面有区别。首先，对任意给定的状态和输入符号，NFA 可有多于一个的可能变迁，NFA 可随意挑选任何一个可用的变迁。当然，某些变迁可能导致阻塞，但只要从起始状态到终结状态存在至少一个变迁序列读完整个输入串，即可称该 NFA 接受这一输入串。

另一区别是 NFA 允许不读入任何输入单词就执行状态变迁，这称为空变迁。从一个状态出发既可以有空变迁也可以有非空变迁，并且 NFA 可随意挑选其中的一个空变迁，或对当前输入符号可用的任一变迁。

不确定的有穷自动机以表格形式表示时，在每一行、列使用目标状态的一个集合，并另加一行表示空变迁。图 3-3 展示了一个简单的 NFA，它识别由所有以 a 开头、并另外包含任意数目的 a 和 d 的串组成的语言。该 NFA 等价于图 3-1 所示的确定 FSA。

图 3-3 中的 NFA 在状态 B 是不确定的，因为从格局 (B, da) 出发，该自动机可转移到以下任意格局之一： (A, a) 、 (B, a) 、 (A, da) 或 (C, da) 。其中，只有前两个格局最终会接受剩下的输入串；状态 A 和状态 C 对沿 d 出发不存在变迁，因而后两个格局都会阻塞。

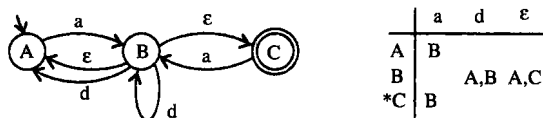


图 3-3 一个不确定的有穷自动机

3.6 将文法转换为自动机

我们用正则文法和正则表达式编写扫描程序的规格说明，剩下的其他工作都是机械的。正则文法与有穷自动机之间保持着一种特殊的关系。根据以下构造规则，可从文法直接构造一个 NFA：

- 字母表仍相同。
- 对文法中的每一非终结符，在 NFA 中创建一个同名的状态；并将目标符号 S 作为起始状态 S 。
- 添加一个新状态，使之成为惟一的终结状态。
- 然后对文法中的每一产生式 $A \rightarrow xB$ ，在 NFA 中构造一个变迁 $\delta(A, x) = B$ 。
- 对文法中的每一产生式 $A \rightarrow x$ ，在 NFA 中构造一个变迁 $\delta(A, x) = F$ ，其中 F 是终结状态。

这样构造的自动机很可能是不确定的，因为正则文法中两个不同的产生式有可能左部有相同的非终结符，且右部有相同的终结符。稍后将介绍如何将 NFA 确定化。

通过将正则表达式转换为一个正则文法，可根据正则表达式构造一个有穷自动机。如果不这样，正则表达式也可按以下算法直接转换为一个执行识别任务的 NFA：定义一个“黑盒”，两端各有一个状态，其中是正则表达式；在左端附加一个起始状态，在右端附加一个停机状态，并从起始状态到黑盒、从黑盒到停机状态各添一条空变迁，如图 3-4 所示；然后对不完整的自动机中每一黑盒应用图 3-5 中的转换规则 F.1~F.5，将其分裂为更小的黑盒，直至不剩任何黑盒。图 3-6 给出了正则表达式 $a(a|d)^*$ 的转换过程。



图 3-4 开始将一个正则表达式转换为一个 NFA

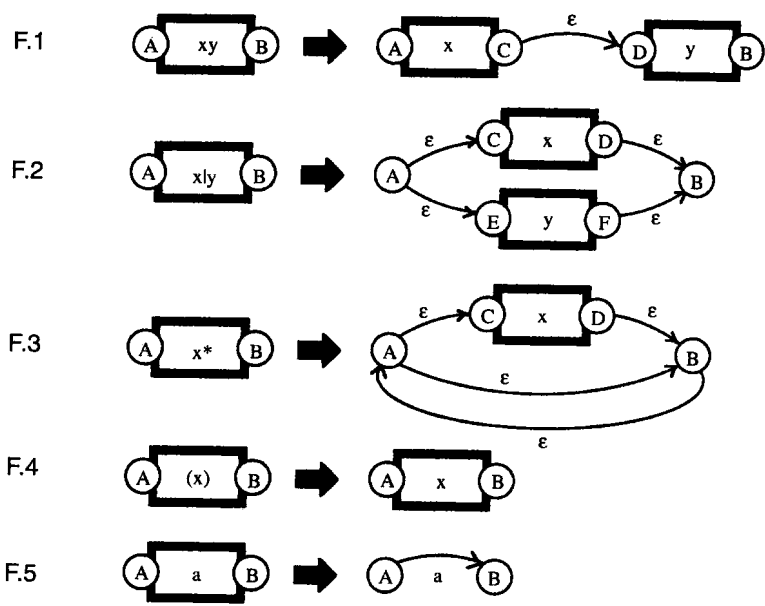


图 3-5 将正则表达式转换为状态变迁的规则

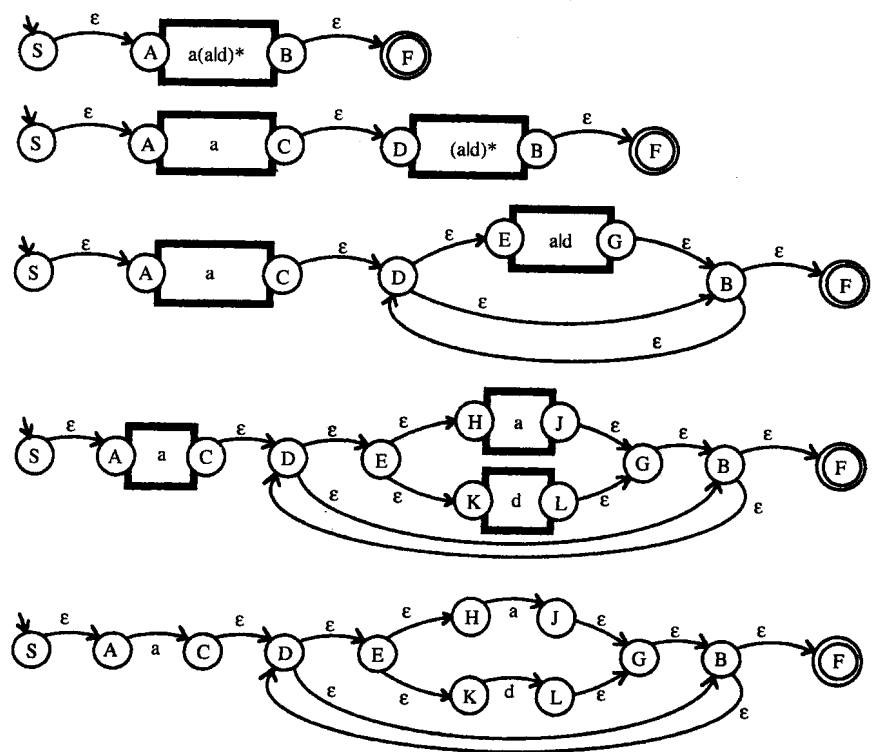


图 3-6 将一个正则表达式转换为一个 FSA

从正则表达式构造一个 NFA 图时，往往有一个很具诱惑性的想法，就是经过仔细推敲后可删除许多空变迁和多余状态。然而正如将正则表达式转换为文法那样，如果遇到嵌套的或连在一起的迭代运算符而未小心处理，则有可能改变了语言。规则 F.3 创建新状态 C 和 D 以及从

A 到 C、从 D 到 B 的空变迁；显而易见，删除这些新状态及其变迁，仅保留从 A 到 B、从 B 到 A 的空变迁，这对许多类似正则表达式 $a(a|d)^*$ 中的简单迭代运算符是可行的，FSA 仍可识别与正则表达式相同的语言，如图 3-7 所示。然而，考虑正则表达式 $(a^*b)^*$ ，其中每一非空串至少以一个 b 结尾。使用错误的规则 F.3 所构造的 FSA 却识别那些仅含 a 的串，如图 3-8 所示，这显然改变了语言。

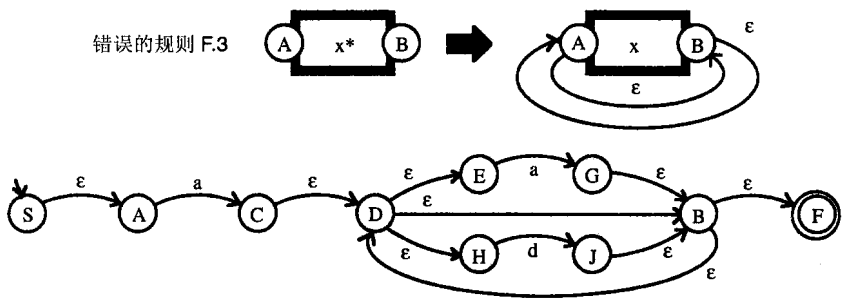


图 3-7 删除空变迁通常看起来是安全的，例如 $a(a|d)^*$ ，但……

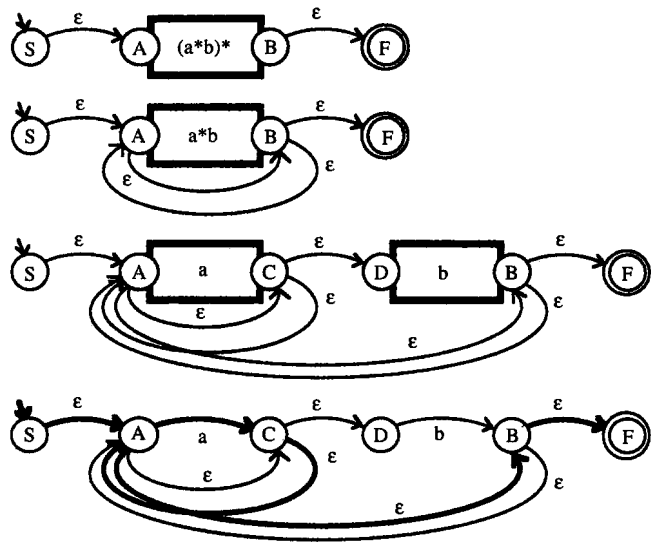


图 3-8 删除空变迁改变了 $(a^*b)^*$ 的语言。单个 a 本是一个不合法的串，其状态变迁为 S-A-C-A-B-F（加粗的弧）

3.7 自动机的转换

如前所述，不确定的自动机在编译程序设计中并不是十分有用。将一个不确定的 NDFA 转换为一个约简的确定 FSA 需要四个步骤，如表 3-5 所示。它们从消除两类空变迁开始。

表 3-5 构造一个确定有穷状态自动机的步骤

1. 将空环路上的每一状态合并为单个状态，从而删除空环路。
2. 将目标状态复制到源状态，从而删除空变迁。
3. 构造一个新的 FSA，其状态代表 NDFA 中状态的集合，从而消除不确定性。
4. 找出状态之间的等价类，从而约简 FSA。

3.7.1 删除空环路

从正则表达式直接构造 NFA 的规则 F.3 引入一个空环路，即两个或多个状态，其间以空变迁相连，从而任一状态可沿空变迁前进到环路中的任何其他状态。显然，一个 FSA 处于任一类状态时，不必读入任何输入，即可前进到环路中的任何其他状态，因而一个空环路中的所有状态都是等价的。因此，将一个空环路中的所有状态可看作单个状态，而不会改变 FSA 所识别的语言。从原环路中任一状态出发的所有变迁成为从新的单个状态出发的变迁，转入原环路中任一状态的所有变迁成为转入单个状态的变迁。所产生的从任一状态到其自身的空变迁（包括从新的单个状态到其自身的任一空变迁）都是冗余的，并可被安全地删除。图 3-9 以图形方式展示了这一步骤，其中含有状态 C 和 D 之间的空变迁（形成含有 2 个状态的空环路），以及 E、B 和 C 之间的空变迁（形成含有 3 个状态的空环路）。由于这两个环路都包含状态 C，所以一个更大的空环路 E-B-C-D-C-E 可安全地合并为单个状态，将它标记为 BCDE。注意，转入任一原状态的所有变迁现转入合并后的状态，从任一原状态出发的所有变迁现改为从合并后的状态出发，空环路本身除外。被合并的状态之间的非空变迁（例如从 D 沿 a 到 B）处理起来与其他变迁一样，然而应注意由于这些变迁的两端均为空环路的状态，该变迁将改为从状态 BCDE（其前身为 D）沿 a 转到自身（其前身为 B）的变迁。

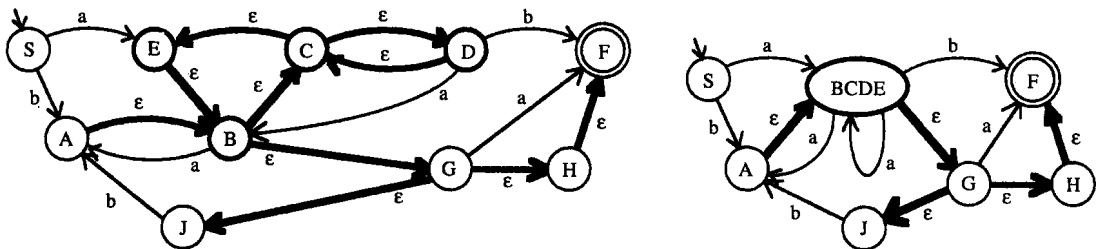


图 3-9 将一个空环路 (E-B-C-D-C-E) 化简为单个状态 (BCDE)

在一个 NFA 的表格表示中更容易找出空环路。表格中所有空变迁用 ϵ 列的入口表示，可为该列的所有入口构造一棵空变迁树，将入口的开始状态连接在一起，如图 3-10b 所示。一旦树中有一条变迁折回树中的已有状态，譬如图 3-10 中从状态 D 回到状态 C，则表示找到一条环路：该回路中的每一状态都属于这一环路，但沿单向空变迁转入或跳出回路的状态均不属于空环路。因而，C-D-C 是一条环路，B-C-E-B 也是一条环路；但从 A 到 B 的空变迁不是环路的一部分，从 B 到 G 的空变迁也不是。此外，尽管从 J 到 A 有一条变迁，但它并不是空变迁，因而它不会产生空环路；我们只需检查表格中的空变迁列。

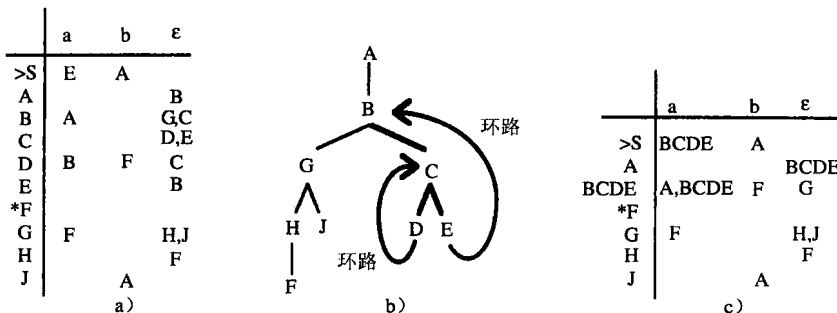


图 3-10 在状态表中找出合并空环路。图 a 中的表是原 NFA；图 b 中的图以单棵树展示了所有的空变迁，其中空环路用粗线表示；图 c 中的表将空环路的四个状态合并为单个状态 BCDE

上述例子中,可独立地将状态 C 和 D 合并为单个状态 CD,然后发现到 CD 属于环路 B-CD-E-B,并再将它合并到单个状态 BCDE;或者也可先将环路 B-C-E-B 合并为单个状态 BCE,然后它成为环路 BCE-D-BCE 的一部分。这两种方法产生的结果均与前述方法相同,即观察到状态 C 同时属于两个环路,因而将两个环路合在一起形成单个空环路。

合并空环路中的状态时,这些状态名字的每次出现被替换为合并后的状态名字。上述例子中,从起始状态 S 沿 a 到 E 的变迁成为从 S 沿 a 到 BCDE 的变迁;保留从 A 到 B 的空变迁作为从 A 到 BCDE 的空变迁。从环路出发的四个变迁(从 B 到 G 的空变迁、从 B 沿 a 到 A、从 D 沿 a 到 B,以及从 D 沿 b 到 F)保留下来,作为从新合并的状态 BCDE 出发的变迁。惟一删除的是从新合并的状态出发并转入该状态自身的空变迁,即原有的空环路变迁。如果原环路中的任一状态是停机状态,这一特征也被复制到合并后的状态,使合并后的状态成为一个停机状态。上述例子不属于这种情况。

3.7.2 删除空变迁

一旦所有空环路通过将其状态合并为单个状态得以删除后,剩下的所有空变迁也可被删除。从任一状态 A 到另一状态 B 的空变迁表明,如果自动机处于状态 A,则它可以执行从 B 出发的任一合法变迁,就好像这些变迁是从 A 出发的那样,因为它不必读进输入即可前进到状态 B (否则会影响其格局)。由于不存在剩余的空环路,反之不成立,即一旦 FSA 执行了从 A 到 B 的空变迁,它无法在不读进输入的情况下回到状态 A。

删除空变迁的方法是将目标状态复制到源状态,即从 B 出发的所有变迁同时也成为从 A 出发的变迁。从图形表示看,这就好像提起 B 的一个副本,同时一起带出它的所有箭头后端复制后的副本,再将 B 的这个副本拉伸到 A 之上,如图 3-11 所示。在此过程中,不复制射入 B 的箭头前端,但如果状态 B 是一个停机状态,则该副本也将携带这一特征。注意,原来从状态 A 出发的变迁仍将保留下来,只是添加了一些从 A 出发的新变迁。

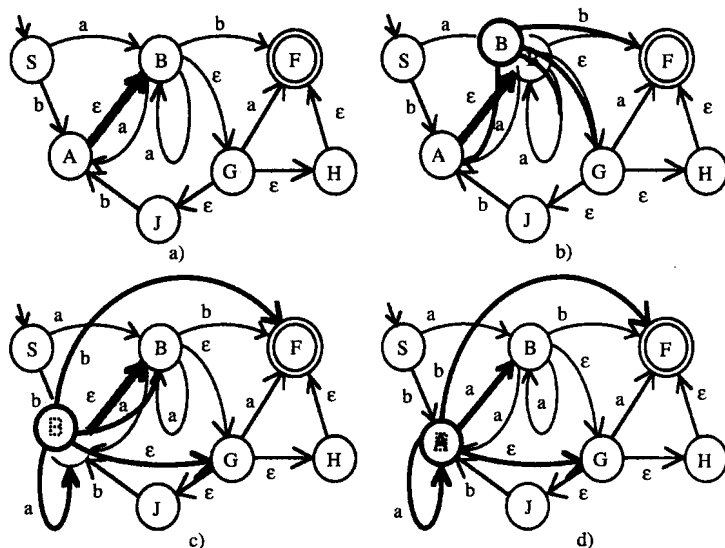


图 3-11 通过复制空变迁的目标状态删除一条空变迁。原 NFA (图 a) 有一条从状态 A 到状态 B 的空变迁,其转换过程是将 B 的一个副本(图 b)“拉伸”到 A 之上(图 c);后端位于 B 的所有弧也被复制,但不复制那些前端在 B 的弧。最后,空变迁被删除(图 d)

图 3-12 中的表格表示更加直观：将 B 那一行的所有东西复制到 A 那一行，包括其他的空变迁，但是引起复制的那条空变迁被删除。如果原来已有从状态 A 沿某一单词 a 出发的变迁，并且从状态 B 也有沿同一单词出发的变迁，则新状态 A 沿单词 a 出发有两个不确定的变迁。如果作为复制源的行是一个停机状态，如图 3-12 的第二步所示，则作为复制目标的那一行也成为一停机状态。注意，复制额外的变迁时只删除引起复制的那条空变迁，空变迁的目标状态仍保留下来。在当前这一步骤，该状态可能是不可访问的，但我们在下一步将解决这一问题。

	a	b	ε				a	b	ε				a	b	ε	
>S	B	A		⊆	>S	B	A			>S	B	A				
A			B		A	A, B	F	G		A	A, B	F	G			
B	A, B	F	G		B	A, B	F	G		B	A, B	F	G			
*F					*F					*F						
G	F		H, J	G	F		H, J	⊆	G	F		H, J				
H			F	H			F		*H							
J				J		A			J			A				
		A														
				a)					b)						c)	

图 3-12 通过复制空变迁的目标状态删除一条空变迁。删除从 A 到 B 的空变迁的方法是将行 B 复制到行 A (图 a)；然后以类似方法将行 F 复制到行 H，以删除从 H 到 F 的空变迁 (图 b)，并令 H 为停机状态 (图 c)

连续应用这一步骤，即可删除所有的空变迁。机灵的读者会注意到，首先从那些以无空变迁转出的状态为转入目标的空变迁入手，手工执行本步骤时可更省事；否则，有些复制工作可能是重复的。这一复制过程总是会终止的，否则将存在一条环路。然而，我们已知上一步骤已删除了所有空环路。

3.7.3 自动机的确定化

一旦删除了所有的空变迁，即可进一步从 NDFA 构造一个确定的 FSA。这一步的做法是定义一个确定的 FSA，它的一个状态代表了原 NDFA 的一个状态集合。FSA 的每一状态记录了对已读入的这一部分输入串的认可。在 NDFA 中，一个给定的部分输入串可到达多个状态中的任意一个；而新的确定 FSA 则对每一可能的部分输入串恰好定义了一个状态。在 FSA 的表格表示中，构造过程简单而直接；图形表示并不会令这一构造过程更可读或减少错误，因而不建议采用图形表示。

开始时建立一个新表格，将起始状态作为该表格的第一个入口；由于仅有一个起始状态，在构造的 FSA 中起始状态仍标记为该状态名。在图 3-13 中，起始状态是状态 A。对添加到新表格中的每一状态 q ，从该状态沿某一特定单词 x 出发的变迁，是确定状态 q 所代表的 NDFA 状态集中的所有状态沿 x 出发的变迁的并集。在这一例子中，A 沿 b 出发有变迁分别到自身、D 和 F，因而这些状态的并集用于命名新的状态：ADF。

当每一个新状态作为一个变迁的目标状态添加到正在构造的表格中时，若该状态还不存在，则将该状态也添加到状态名称行的列表中。在图 3-13 的例子中，起始状态将状态 B 和 ADF 添加到列表中；类似地，状态 B 有变迁转到 C 和 F，因而添加了新状态 CF。

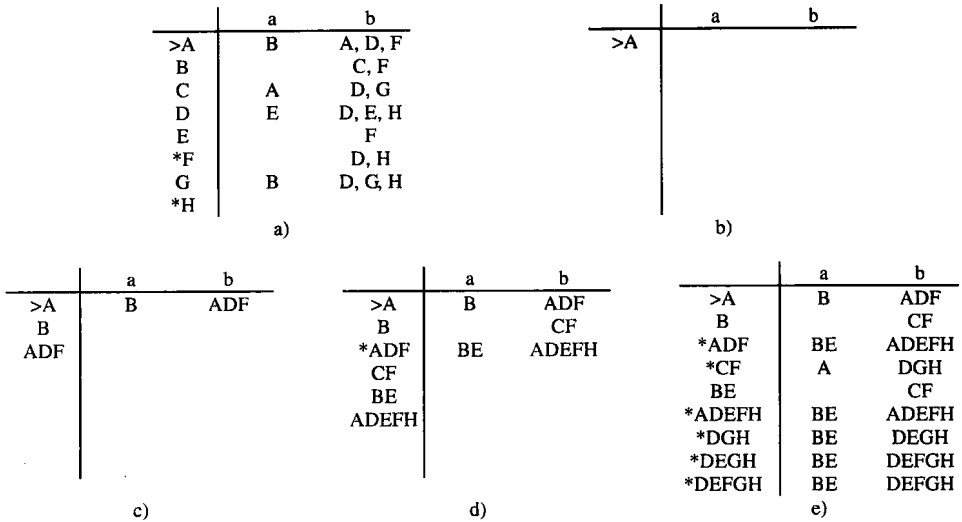


图 3-13 从 NFA (图 a) 构造一个确定的 FSA。从起始状态入手 (图 b), 并填写其变迁。对作为变迁目标的每一新状态, 将它添加到状态名字中 (图 c)。在 NFA 中作为一条变迁的目标的每一状态集命名了一个新状态 (图 d), 除非该状态已有命名。当不再有新状态加入时, 这一过程结束 (图 e)

状态 ADF 是状态 A、D 和 F 的并集。由于 F 是停机状态, 因而 ADF 也是停机状态。在 NFA 中, 状态 A 沿 *a* 转到 B、而 D 沿 *a* 转到 E, 故其并集转到名为 BE 的状态; F 无沿 *a* 出发的变迁, 因而不会添加任何东西。类似地, 对输入单词 *b*, ADF 转到 {A, D, F} (从 A 出发)、{E, D, H} (从 D 出发) 和 {D, H} (从 F 出发) 的并集, 即集合 {A, D, E, F, H}, 记为 ADEFH。

处理状态 BE 时, 会发现从状态 B 和 E 的并集沿 *b* 出发的变迁产生了状态 CF, 而这一状态已出现在新的状态集中。从状态 ADEFH 出发的变迁也不会添加任何新的状态。因而, 得到的确定 FSA 有 9 个状态。

3.7.4 自动机的约简

从 NFA 构造一个确定的 FSA 的最后一步是将它约简, 使之状态数目最少。尽管这一步实际上是 FSA 确定化工作的一部分, 但这种有益的想法却往往有助于将扫描程序压缩到一个易于管理的规模。约简 FSA 的方法是定义状态的等价类; 其中, 两个状态被指派到同一等价类的条件是 FSA 在这两个状态中表现出同样的行为, 即对每一个给定的部分串, 如果 FSA 从这两个状态之一出发可识别该串, 则它从另一状态出发也将识别该串, 反之亦然。开始时, 我们将状态分为两个等价类, 其中惟一的区分是状态是否是终结的, 如图 3-14 所示。在这两个等价类中, 停机状态识别空串, 而其他状态无法识别。

只要可找出同一等价类中的两个状态沿同一单词出发的变迁转入两个不同的等价类, 这就意味着在该单词作为下一输入的格局中, FSA 对剩余输入串的反应有所不同, 至少存在一个子串使得一个状态最终接受它, 而另一状态最终拒绝它, 因而它们分属不同的等价类。上一例子在第一次细分后, 所有停机状态均沿 *a* 转入另一等价类, 均沿 *b* 转入其自身, 因而没有区别。然而, 非停机等价类中状态 A 沿 *a* 转入状态 B (位于本等价类中), 而状态 B 和 BE 则会阻塞; 尽管这三个状态沿 *b* 到达停机等价类中的一个状态, 但它们沿 *a* 出发的变迁有所区别。

	a	b		a	b
>A	B	ADF	>A	B	ADF
B		CF	B		CF
BE		CF	BE		CF
*CF	A	DGH	*CF	A	DGH
*ADF	BE	ADEFH	*ADF	BE	ADEFH
*ADEFH	BE	ADEFH	*ADEFH	BE	ADEFH
*DGH	BE	DEGH	*DGH	BE	DEGH
*DEGH	BE	DEFGH	*DEGH	BE	DEFGH
*DEFGH	BE	DEFGH	*DEFGH	BE	DEFGH
a)			b)		

	a	b		a	b
>A	B	ADF	>A	B	D
B		CF	B		C
BE		CF	*C	A	D
*CF	A	DGH	*D	B	D
*ADF	BE	ADEFH	d)		
*ADEFH	BE	ADEFH			
*DGH	BE	DEGH			
*DEGH	BE	DEFGH			
*DEFGH	BE	DEFGH			
c)					

图 3-14 约简一个确定的 FSA。先将状态划分为两个等价类：停机与不停机（图 a）。在任一等价类中，若对某一特定输入某些状态阻塞而另一些状态不阻塞，则可进一步细分该等价类（图 b）。若同一等价类中的状态沿同一输入符号存在变迁转入两个不同的等价类，则这些状态亦可细分（图 c）。不存在进一步细分时，每一等价类就是约简后的 FSA 中的一个状态（图 d）

在此细分之后，可观察到停机等价类中的状态 CF 沿 a 转入包含状态 A 的等价类，而所有其他的停机状态则转入此时位于另一等价类的状态 BE，因而找出状态 CF 并将其单独作为一个等价类。所有等价类的细分都基于等价类边界之间的变迁。

当不再有等价类可进一步细分时，称 FSA 是约简的。图 3-14 所示的例子中，未约简的表格中最后 5 个状态沿输入单词 b 前进到 3 个不同状态，但这 3 个状态均在同一等价类中，基于这一点找不出它们之间的任何区别。因而可得出结论，约简的 FSA 有 4 个状态。为清晰起见，图 3-14 最后一幅图中重新命名了状态。

为一个扫描程序花费工夫去约简其 FSA 未必是合算的。等价状态通常是用于定义一个扫描程序的正则表达式中的选择运算所产生的结果。像 Modula-2 这类典型程序设计语言的扫描程序，可能在总数为 50~200 个的状态中会产生十多个等价状态。

3.8 将自动机转换为文法

尽管在构造一个编译程序时很少有理由这样做，但有意思的是有穷状态自动机可以很简单地转换为一个等价的文法。可反过来直接应用从文法构造一个 FSA 的规则。对每一变迁：

$$\delta(A, x) = B$$

可在文法中写出一条产生式：

$$A \rightarrow xB$$

对每一停机状态 F，再在文法中添加一条产生式：

$$F \rightarrow \epsilon$$

将这一构造过程应用到图 3-14 中约简的 FSA，可得如下右线性文法的 9 条产生式：

$$A \rightarrow aB$$

$$C \rightarrow aA$$

$$D \rightarrow aB$$

$A \rightarrow b D$
 $B \rightarrow b C$

$C \rightarrow b D$
 $C \rightarrow \epsilon$

$D \rightarrow b D$
 $D \rightarrow \epsilon$

用规则 R.5 将 C 和 D 的空产生式回代对它们的引用，可得正则文法：

$G = (\{ a, b \}, \{ A, B, C, D \}, P, A)$

其中，P 是如下产生式集合：

$A \rightarrow a B$
 $A \rightarrow b D$
 $A \rightarrow b$
 $B \rightarrow b C$

$C \rightarrow a A$
 $C \rightarrow b D$
 $C \rightarrow b$
 $B \rightarrow b$

$D \rightarrow a B$
 $D \rightarrow b D$
 $D \rightarrow b$

因为已有方法从正则文法构造一个正则表达式，因而有可能在正则文法、正则表达式、确定的有穷状态自动机三者中从其中一种形式转换为另一种等价的形式。通常我们从正则文法或正则表达式入手，根据它构造一个 FSA。

3.9 左线性文法

到目前为止，我们仅关注右线性文法和右正则文法，但偶尔也有必要处理一个左线性文法，通常是将其转换为一个等价的右线性文法。完成这一工作的最简单算法是应用表 3-4 中规则 R.1~R.3 的镜像，先将左线性文法转换为一个正则表达式，然后（若有必要）按常规方法将正则表达式转回一个右线性文法或正则文法。

这些镜像规则意识到，在一个左线性文法中，每一产生式右部的非终结符均出现在左端；表 3-6 简要总结了这些规则。注意，规则 L.2 仅用于将左线性文法转换为正则表达式；从嵌套的迭代运算符转换为一个文法时，需要另一个非终结符，如表 3-3 所示。

表 3-6 将左线性文法转换为正则表达式

规则#	文法的产生式	正则表达式的产生式
L.1	$A \rightarrow B x \quad B \rightarrow y$	$A \rightarrow y x$
L.2	$A \rightarrow A x \quad A \rightarrow y$	$A \rightarrow y x^*$
L.3	$A \rightarrow x \quad A \rightarrow y$	$A \rightarrow x y$

3.10 在计算机上实现有穷状态自动机

大多数编译程序的实现是一个传统计算机上的程序。尽管我们采用编译程序生成工具从文法和正则表达式直接构造大多数编译程序，但了解如何根据同一文法以手工方式构建一个扫描程序也会带来启示。实际上，有必要学习的只是如何将将有穷状态自动机实现为一个计算机程序，因为从文法或正则表达式转换为 FSA 的每一步骤都是已知的。

计算机是一个带输出的有穷状态自动机。现在暂时忽略输出，我们可观察到输入“带”实际上就是终端键盘（或是当前通过本地或远程连接的所有输入键盘的集合）。为简化起见，假设只有单个本地的键盘。FSA 无法在输入带上回头，计算机亦无法“回卷”或退回键盘的输入；一旦从键盘读入了一个字符，它就不再可用。

作为一个 FSA 的计算机，其字母表 Σ 是在键盘上可键入的所有字符的集合，再加上一个特殊符号“ \circ ”表示输入条件“尚未键入任何东西”。由于计算机远快于在键盘上击键的人，由键盘表示的输入带的大部分将由符号“ \circ ”填充。当操作人员输入 **The quick brown fox ...**

时，计算机将看到类似如下的输入带：

```

ooooooooTooooooooh°°e°°°  oooooq°°°°u°°°°i°°°°°c°°°°k°°°°
oooooooo°p°°°°°°°°r°°°°°o°°°°°w°°°°n°°°  o°°°°f°°°°°o°°°°°°°°°°°°x°°°°

```

计算机中所有寄存器、内存、联机磁盘存储器等均可用于状态的编码，这导致可能的状态数目多得难以置信：不计寄存器和磁盘存储器，一台内存为 1MB 的计算机具有 $2^{8\ 388\ 608}$ （大约相当于 $10^{2\ 500\ 000}$ ）个可能的状态。这相当于在一个 1 后面有超过 200 万个 0，是比宇宙中已知的亚原子粒子数目还多出很多倍的惊人天文数字。因而，仅仅以有穷状态自动机的观点思考一台普通的计算机并不会带来什么特别的指导意义。

然而，可为计算机编程以模仿一个小得多的 FSA。一台运行不带输出的非递归计算机程序的计算机也是一个有穷状态自动机。程序中的所有变量，加上程序计数器（硬件中的当前指令地址），一起编码为状态；常量和可执行的代码合在一起表示变迁规则；起始状态是程序首次开始执行第一条指令时的状态。我们时常听到程序的变量在程序启动时有“未定义的”内容这种说法，事实上这意味着这些内容将由内存中的上一程序定义，或意味着程序员未被告知这些内容是什么。在某种程度上这一说法是正确的，因而我们可能必须留意到含有未初始化变量的某一特定程序的起始状态会略有不同，这取决于在运行程序之前内存中有什么东西。

我们用更少的状态信息构造一个扫描程序的 FSA：一个枚举类型的变量。枚举类型列出了 FSA 的所有可能的状态，变量则恰好是这些状态的编码，除此之外别无含义。为简便起见，我们根据 FSA 的表格表示构造这一程序，并用程序中的一个数组显式地保存这一表格。该数组形式上是一个变量，但由于它被初始化后在程序执行过程中不会改变，因而它并不表示状态。我们使用一个布尔变量标记 FSA 的终止，当程序正在运行时它总为 **TRUE**，因而它也没有表示状态信息。一个临时变量保存了当前的输入符号，但其长度仅够用于检索状态表；尽管它其实是状态信息的一部分，但该程序并未按此方式使用它。

采用 Modula-2 语言编写的程序如代码清单 3.1 所示，其中给出了图 3-1 的 FSA 的必要组成部分。枚举类型 **State** 指出 FSA 的所有可能的状态，即 **AA** 和 **BB**。当前状态变量 **CurrentState** 属于类型 **State**，表示 FSA 的全部状态。数组 **NextState** 是状态变迁规则的编码。从模块 **ReadInputTape** 导入的枚举类型 **Alphabet** 指明所有的输入符号。停机状态显式地列在常量集合 **HaltStates** 中。

代码清单 3.1 一个简单的有穷自动机的 Modula-2 实现

```

MODULE ReadInputTape;
EXPORT
  Alphabet, atEnd, nextInput;
TYPE
  Alphabet = (a, d);
PROCEDURE atEnd: BOOLEAN;
PROCEDURE nextInput: Alphabet;
END ReadInputTape.

MODULE FSA;
FROM ReadInputTape IMPORT
  Alphabet, atEnd, nextInput;
FROM InOut IMPORT
  WriteString;

```

```

TYPE
    State = (AA, BB);
CONST
    HaltStates = State{BB};
VAR
    CurrentState: State;
    notDone: BOOLEAN;
    theInputToken: Alphabet;
    NextState: ARRAY State, Alphabet OF RECORD
        isValid: BOOLEAN;
        theState: State;
    END;

PROCEDURE InitializeTable;
BEGIN
    NextState[AA, a].isValid := TRUE;
    NextState[AA, d].isValid := FALSE;
    NextState[BB, a].isValid := TRUE;
    NextState[BB, d].isValid := TRUE;
    NextState[AA, a].theState := BB;
    NextState[BB, a].theState := BB;
    NextState[BB, d].theState := BB;
END InitializeTable;

BEGIN (* FSA 程序 *)
    InitializeTable;
    CurrentState := AA;
    notDone := TRUE;
    WHILE notDone DO
        IF atEnd() THEN
            notDone := FALSE;
            IF CurrentState IN HaltStates THEN
                WriteString('Accept')
            ELSE
                WriteString('Reject')
            END
        ELSE
            theInputToken := nextInput();
            notDone := NextState[CurrentState, theInputToken].isValid;
            IF notDone THEN
                CurrentState := NextState[CurrentState, theInputToken].theState
            ELSE
                WriteString('Reject')
            END
        END (* IF atEnd *)
    END (* WHILE notDone *)
END FSA.

```

该程序使用了两个函数过程，但并未声明它们，其目的分别是输入带读入下一个符号，以及确定输入是否已到达结尾。从程序中可看出，它们与类型 **Alphabet** 一起从模块 **ReadInputTape** 导入。

该程序有三种可能的终止方式。如果在表 **NextState** 中，当前状态和输入单词没有合法的变迁，FSA 阻塞并拒绝输入串；如果到达输入串结尾（函数 **atEnd** 返回 **TRUE**）且 FSA 处

于停机状态，则接受输入串；如果到达输入串结尾而 FSA 不处于停机状态，则拒绝输入串。程序中的主循环测试这三种终止条件，如果不成立则沿输入符号前进到下一状态。每次经过主循环（最后一次除外）都读入恰好一个输入符号，并恰好前进一个状态变迁。

代码清单 3.1 并不是以程序设计语言编写 FSA 的最有效实现。由于我们通常（对扫描程序而言则几乎总是）关注执行效率，因而我们寻找改进其性能的途径，同时又不降低设计的严密性。第一种同时也是最显然的方法是在变量 **CurrentState** 中包含非法变迁的信息，从而只需一次查表。具体做法是定义一个或多个状态表示出错状态，并在表中删除记录的定义（因为不再需要布尔类型的字段）；如果 FSA 进入了出错状态，这表明形式化 FSA 已阻塞。因此，主循环可改进为类似如下的代码：

```
WHILE (CurrentState <> ErrorState) AND NOT atEnd() DO
    CurrentState := NextState[CurrentState, nextInput()]
END; (* WHILE *)
IF CurrentState = ErrorState THEN
    WriteString('Reject')
ELSIF CurrentState IN HaltStates THEN
    WriteString('Accept')
ELSE
    WriteString('Reject')
END
```

在循环的每次迭代中，测试终止条件是不可避免的（尽管稍后将看到这仍可作少许改进），从计算的观点看，查表（特别是二维表）是相当费时的。在某些计算机上，可将这个表编码为程序代码，以大幅度改进性能。这使得程序更大且更难构造，但节省了执行时间。代码清单 3.2 展示了改进的结果。

代码清单 3.2 将有穷自动机的变迁编码为程序代码

```
MODULE FSA;
FROM ReadInputTape IMPORT
    Alphabet, atEnd, nextInput;
FROM InOut IMPORT
    WriteString;
TYPE
    State = (AA, BB, ErrorState);
CONST
    HaltStates = State{BB};
VAR
    CurrentState: State;
    notDone: BOOLEAN; (* 不再需要 *)
    theInputToken: Alphabet;
BEGIN (* FSA 程序 *)
    CurrentState := AA;
    WHILE (CurrentState <> ErrorState) AND NOT atEnd() DO
        theInputToken := nextInput(); (* 读入输入带 *)
        CASE CurrentState OF
            AA: (* 状态 AA 的分支情况 *)
                CASE theInputToken OF
                    a:
                        CurrentState := BB |
```

```

        d:
            CurrentState := ErrorState
        END |
    BB:                                     (* 状态 BB 的分支情况 *)
        CASE theInputToken OF
            a, d:
                CurrentState := BB
            END
        END (* CASE CurrentState *)
    END; (* WHILE *)
    IF CurrentState = ErrorState THEN
        WriteString('Reject')
    ELSIF CurrentState IN HaltStates THEN
        WriteString('Accept')
    ELSE
        WriteString('Reject')
    END (* IF Accept/Reject *)
END FSA.

```

CASE 语句相当于机器语言的查表，但这有两个优点：首先，**CASE** 表是一维的，通常效率更高；其次，许多分支情况退化为代码，会比写成 **IF-THEN-ELSE** 语句更高效。例如，状态 **BB** 的代码根本不做任何判断（对所有输入仍留在状态 **BB**），因而它可替换为空语句；类似地，状态 **AA** 只需对输入单词测试一次，判断它是否单词 **a**：

```

CASE CurrentState OF
    AA:                                     (* 状态 AA 的分支情况 *)
        IF theInputToken = a THEN
            CurrentState := BB
        ELSE
            CurrentState := ErrorState
        END |
    BB:                                     (* 状态 BB 不做任何事情 *)
END (* CASE CurrentState *)

```

FSA 的状态本身可编码为程序计数器，以取得最佳运行速度。结果是程序代码不仅表示了状态变迁表，还表示了 FSA 的状态，从而可消除测试状态变量的 **CASE** 语句，代之以复制的代码。程序又再变得更大、但更快，这是计算机程序设计中用空间换取时间（反之亦然）的常见折中。代码清单 3.3 展示了这一改进。注意，这一改进丝毫没有危及 FSA 定义的形式化正确性：输入字母表仍是相同的；状态名字仍是完全可识别的（只是此时用注释标记了代表该状态的代码段）；变迁表也像代码清单 3.2 一样编码在程序代码中。

代码清单 3.3 将有穷自动机的状态编码为程序计数器

```

MODULE FSA;
FROM ReadInputTape IMPORT
    Alphabet, atEnd, nextInput;
FROM InOut IMPORT
    WriteString;
TYPE
    State = (AA, BB, ErrorState);
CONST
    HaltStates = State{BB};

```

```

VAR
  CurrentState: State;
  notDone: BOOLEAN;
  theInputToken: Alphabet;
BEGIN
  IF atEnd() THEN
    WriteString('Reject')
  ELSIF nextInput() <> a THEN
    WriteString('Reject')
  ELSE
    WHILE NOT atEnd() DO
      IF nextInput() IN Alphabet{a, d} THEN
        END
      END; (* WHILE *)
      WriteString('Accept')
    END
  END FSA.

```

(* 不再需要 *)
 (* 不再需要 *)
 (* FSA 程序 *)
 (* 从状态 AA 开始 *)
 (* 状态 AA 不是停机状态 *)
 (* 在状态 AA 读入输入带 *)
 (* 错误输入导致阻塞 *)
 (* 前进到状态 BB *)
 (* 仅在状态 BB 读入输入 *)
 (* 状态 BB 结束 *)

将状态编码为程序计数器的一个难点是一个任意的 FSA 需要 **GOTO** 语句从一个状态转入另一状态。这对大多数计算机的机器语言都不成问题，但现代程序设计语言不鼓励在源程序中使用 **GOTO** 语句。上述的简单例子很幸运，仅用一个 **IF-THEN-ELSE** 结构即可实现，但在一般情况下这是不可能的。此外，以手工方式正确地编写代码实现这样的 FSA 是相当困难的，我们往往只寄望于一个“扫描程序—编译程序”完成此项任务。“编译程序—编译程序”的输入源语言是一个文法或正则表达式（并非像 Modula-2 这类程序设计语言的代码），因而使用 **GOTO** 语句不成问题。

乍看起来，代码清单 3.3 好像只是直接改写了定义语言的正则表达式，在这一例子中的确如此。然而在更普遍的情况下，识别一个简单正则表达式所定义的语言的确定 FSA 可能并不简单，并且程序的构造过程也远远没有这么直接。例如，一个识别正则表达式 $(a|b)^*aaba$ 所定义的语言的识别程序中，用一个简单的 **WHILE** 循环作为程序的开头显然是不够的。为该正则表达式构造一个正确的识别程序将留作练习。

3.11 扫描程序的特殊实现问题

涉及真实的扫描程序实现时，会遇到三个未正式强调的特殊问题。

3.11.1 输入字母表的大小

第一个问题与输入字母表的大小有关。在一个典型的扫描程序 FSA 中，状态的数目约为 60 个左右；如果将整个 ASCII 字母表用作输入，将有 128 个字符。结果是扫描程序的表格将有 8 000 个入口，直接的代码实现将会更大。这对一台大型机可能是可接受的，但当今许多更小的计算机却不可接受。

解决这一问题的常见途径是定义字符的等价类，从而所有数字处理起来好像是输入字母表中的单个符号一样；所有字母（字母 **E** 除外，它用在实数常量的表示中）作为另一类；语言中未有特定含义的所有特殊符号（例如重音符“`”很少指定为语言中的字符）也作为字母表中的单个符号。这通常可将字母表压缩到不足原大小的一半，其代价是一次小的查表。

图 3-1 中 FSA 的字母表仅有 2 个符号： a 和 d ，如果将它们分别理解为字母和数字的等价

类集合，易见该机器识别 Modula-2 语言（或 Pascal 语言）的所有标识符：标识符必须以字母开头，然后可以有任意数目的字母和数字。将 ASCII 字符集转换为一个该 FSA 适用的字母表，则该表将有 128 个单字节的入口，每一入口要么包含枚举常量 a 或 d ，要么包含某一其他符号（将被识别为错误）。

3.11.2 扫描程序自动机中的停机状态

第二个实现问题势必涉及识别单词串的结尾。迄今为止，有穷状态自动机的每一实现都调用了未定义的函数 `atEnd()` 以确定输入串的结尾。然而在实际应用中，这样的函数并不存在，扫描程序自身有义务确定一个单词的结尾。

考虑从正则表达式 d^+d^+ 构建的一个扫描程序，如果它读进输入串 12435 时会发生什么？如果扫描程序期望找出两个整数常量，第一个结束和第二个开始的位置在哪里？可能是 124 后面跟着 35，或 12 后面跟着 435，亦或只是单个整数 12435。Pascal 语言和 Modula-2 语言不允许出现这种情况，要求相邻的数字和标识符之间有一个或多个空格。但问题仍存在：FSA 在何处停止读进输入字符并停机？

这一问题仅限于在一个有转出变迁的停机状态中如何识别输入串的结尾，通常有两种解决方案。一种解决方案是每次读入时，将下一输入符号放到一个缓冲变量中，然后在每一个有转出变迁的停机状态查看缓冲变量，检查下一符号是否能使这些变迁中的某一个发生。如果能则读入符号并执行变迁，否则停机。

另一解决方案更通用，完全忽略停机状态，FSA 一直运行，直至遇到某一输入字母后阻塞。若此时 FSA 处于一个终结状态，则停机并接受该输入串；否则，拒绝输入串。在这两种情况下，均须保存导致阻塞的字符，并用它启动 FSA 识别下一单词。

这两种方法均未能检测错误的串 123abc，而是报告找到一个数字，后面还接着一个标识符。

3.11.3 过滤空格与注释

在大多数语言中，注释和空格并不是单词，扫描程序在扫描时会从源程序文本中删除它们，而不会将它们报告给分析程序。在 Pascal 语言和 Modula-2 语言中，注释可出现在“允许空格字符出现的任何地方”（在字符串常量中除外），并且具有与空格字符相同的语义值。

可为注释编写一个正则表达式或正则文法，但识别出的注释的语义动作是空的。类似地，也可空格字符编写简单的正则表达式或正则文法。通过将空格字符的正则表达式附加到一个单词的复合正则表达式的最前面，扫描程序可显式地支持空格并删除这些空格。考虑如下正则表达式定义的 Modula-2 单词的子集：

T.1 $a(a|d)^+ \mid dd^+ \mid "(" \mid ")" \mid "+" \mid "-" \mid "**"$

该正则表达式定义了一个由标识符、正整数常量、圆括号以及整数加法、减法和乘法运算符等组成的集合。加上注释和空格的规格说明后，结果如下所示：

T.2 $(""|c)^+(a(a|d)^+ \mid dd^+ \mid "(" \mid ")" \mid "+" \mid "-" \mid "**")$

其中， c 是注释的正则表达式。因而，任意数目的空格字符和注释可以按任意次序出现在任何其他单个单词之前。

对于 Pascal 和 Modula-2 这类语言，在单词的文法或正则表达式中表达这种设计是很重要的，因为注释分隔符采用了与其他合法单词相同的字符，这会对 FSA 的实现带来重大影响。根据含有注释定义的正则表达式构造一个确定的 FSA 时，识别左括号的新停机状态会与识别其他

单字符单词的停机状态有所不同，如图 3-15 所示。这一新的状态沿星号出发有一条变迁，如果沿该变迁走下去，会在另一单词（可能是不同的单词）之前组成一个注释。

3.11.4 单词的输出

扫描程序实现中的最后一个问题更为重要，并且仅在传递输出时才涉及该问题。一个扫描程序不仅读入并识别其输入串，而且还为它识别的每一合法输入串产生输出的单词。一个形式化的有穷状态自动机没有输出，解决这一问题的办法是扩展 FSA

的定义, 为状态变迁添加语义动作, 定义与变迁相关联的 0 个或多个语义动作列表作为每一变迁的组成部分。尽管在理论上对语义动作的功能没有限制, 但实际上它们仅限于收集一个标识符或字符串单词中的字符, 并为该扫描程序语言中识别出的每一个串输出一个单词符号。

再次考虑 T.2 中定义的 **Modula-2** 语言的单词的子集，它定义了一个由标识符、正整数常量、圆括号以及整数加法、减法和乘法运算符等组成的集合。扫描程序会找出这 7 类单词中的每一种，并作为分析程序输入字母表中的惟一单词传递给分析程序。这可能作为一个枚举类型中的值：

TYPE

```
Token = (id, num, left, right, plus, minus, star);
```

将这些单词的值传递给分析程序的语义动作可添加到正则表达式中的合适位置。将它们写进正则表达式时，用元符号“方括号”括住，以区别于字母表中的符号。结果类似于：

$$a[\text{id}](a|d)^*|d[\text{num}]d^*| "("[\text{left}]| ")"[\text{right}]| "+"[\text{plus}]| "-"[\text{minus}]| "*"[\text{star}]$$

在同一语言的（右线性）文法中，亦可用类似方式写进相同的语义动作：

$$S \rightarrow aA \quad [\text{Tok} = \text{id}] \quad A \rightarrow dA$$
$$S \rightarrow d N \quad [\text{Tok} = \text{num}] \quad A \rightarrow a A$$
$$S \rightarrow "(" \quad [\text{Tok} = \text{left}] \quad A \rightarrow \epsilon$$
$$S \rightarrow ") " \quad [\text{Tok} = \text{right}] \quad N \rightarrow d N$$

$S \rightarrow "+" \quad [\text{Tok} = \text{plus}] \quad N \rightarrow \epsilon$

$S \rightarrow "*" \quad [\text{Tok} = \text{star}] \quad S \rightarrow "-"$

[Tok = minus]

根据文法或正则表达式构造的一个扫描程序 FSA 在其变迁规则中保留了语义动作,如图 3-16 所示。在 Modula-2 这类程序设计语言中,这些语义动作的组成也就是在区分每一类单词的变迁上多写一行代码而已。在基于表的实现中,开设第三个字段将语义动作编码为在每一变迁上待执行的动作。如果表的入口是一个过程的指针,则当指针不为 **NIL** 时该过程会被调用;如果表的入口是一个枚举类型的值,则可用一条 **CASE** 语句选择合适的语义动作;如果表的入口是类型 **TOKEN** 的一个值,可直接将它赋值给输出的单词变量,如代码清单 3.4 所示。

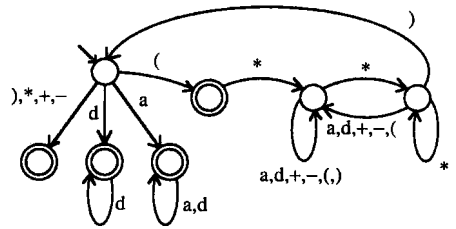


图 3-15 在 FSA 中将注释与其他单词分开

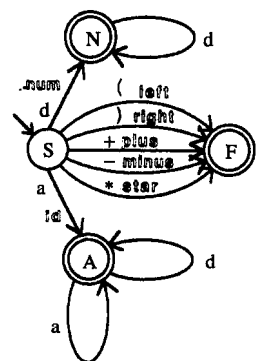


图 3-16 FSA 中的语义动作（用空心字体表示）

代码清单 3.4 在扫描程序代码中语义动作编码的三种方法

```

WHILE notDone DO                                (* 直接赋值给输出单词 *)
    ...
    theInputToken := nextInput();
    NextToken := NextState[CurrentState, theInputToken].SemanticAction;
    CurrentState := NextState[CurrentState, theInputToken].theState;
    ...

WHILE notDone DO                                (* 根据适当语义编写的内嵌代码 *)
    ...
    theInputToken := nextInput();
    CASE NextState[CurrentState, theInputToken].SemanticAction OF
        id:
            Stuff(theInputToken);                (* 参阅字符串表的有关章节 *)
            NextToken := id |
        num:
            Value := Value * 10 + ORD(theInputToken) - ORD('0');
            NextToken := num |
        plus:
            NextToken := plus |
        ...
    END; (* CASE SemanticAction *)
    CurrentState := NextState[CurrentState, theInputToken].theState;
    ...

WHILE notDone DO                                (* 使用表中的过程指针 *)
    ...
    theInputToken := nextInput();
    SemProcPtr := NextState[CurrentState, theInputToken].SemanticAction;
    IF SemProcPtr <> NIL THEN
        SemProcPtr(theInputToken)                (* 调用该过程 *)
    END;
    CurrentState := NextState[CurrentState, theInputToken].theState;
    ...

```

据此，我们有一种确定的方法将语义动作附加到扫描程序的定义中。按本书的习惯写法，我们用方括号括住语义动作，从而将语义动作与纯文法区别开来。第4章介绍的编译程序生成工具识别语义动作的这一语法，并构建合适的目标代码处理这些语义；其中所用的语义动作的特定语法与上述例子略有不同，但这一概念在本质上是相同的。编译程序生成工具的其他实现则采用了不同的表示法以取得同样的效果，但这些区别也主要体现在表示法上。

注意，扫描程序的自动机在任何一种情况下，都从其输入流中读入（并识别）恰好一个单词，然后“停机”。扫描程序停机意味着它停止读进输入，并将一个单词的值返回给分析程序，而并不是计算机停止运行。扫描程序识别的语言是所有可能的单词的集合，每次找出一个串，因而在一次给定的调用中，扫描程序不会向前越过它识别出的单个输入单词。当分析程序准备好处理另一单词时，它将重新启动扫描程序进入它的起始状态，并在扫描程序停机时接收到另一单词。

3.12 字符串表的实现

当扫描程序从任一给定标识符的特定拼写中抽取出单词的 `id` 时，必须保留这些拼写，以

备稍后用于报错信息、内存映像表、模块间链接等。要求分析程序来处理这些细节是低效的，因为对标识符的大多数引用并不会像关注它在上下文中的惟一性那样关注它的拼写。编译程序需要接触的输入文本中各个字符的次数越少，它就运行得越快。因而，扫描程序有义务保存标识符和字符串常量的拼写，并且仅在有请求时才让它们可用。这一需求由一个名为字符串表的数据结构实现。

一个字符串表本质上是一个大型的压缩字符数组，其中每一标识符和字符串常量按首尾相接方式加注标签并存储。标签可能包含长度或终止符代码，也可能含有搜索队列中下一项目的链接。变长标识符处理起来不会有浪费，所以再也没有必要限制源语言中标识符的长度。字符串在表中的位置被传递给分析程序和约束程序，作为对标识符的惟一引用；因而，后续对标识符的相等比较只需基于一个简单的整数下标即可完成，而不是费时的变长字符串比较。

当 FSA 读入一字符时，扫描程序中标识符对应的语义动作代码将每一个新的字符保存到表中的下一可用入口。如果随后发现该标识符在表中已存在，则只要忽略新的入口即可快速回收空间；如果它真的是一个新的标识符，则指向表尾的指针前进到新字符串的结尾，该字符串被链接到表中。

3.12.1 基于线性查找的实现

在一个最简单的实现（参阅代码清单3.5）中，匹配每一个新字符串都需要查找整张表。由于新字符串在表尾插入，查找结果总会以匹配成功而告终，可能匹配到的就是新字符串本身。

代码清单 3.5 以线性查找方式实现的字符串表

```
(* 过程 UniqueIdent 用到的全局声明 *)
CONST                                     (* 字符串结束代码 *)
    EOS = 0C;                           (* 每一字符串以 EOS 为终结 *)
VAR
    StringTable: ARRAY [1 .. MaxStrings] OF CHAR;
    TableEnd: INTEGER;                   (* 新符号的起点、旧表的结尾 *)
    NextStringTableEntry: INTEGER;      (* 正在处理的新符号的结尾 *)

PROCEDURE UniqueIdent(): INTEGER;
VAR
    SearchAt, NewSymbol, StringStart: INTEGER;
BEGIN
    SearchAt := 1;
    NewSymbol := TableEnd;                (* 新字符串已保存在表尾 *)
    StringTable[NextStringTableEntry] := EOS;    (* 新字符串的终结 *)
    WHILE SearchAt < NewSymbol DO              (* 不与其自身进行字符串比较 *)
        StringStart := SearchAt;
        WHILE (StringTable[SearchAt] = StringTable[NewSymbol])
            AND (StringTable[SearchAt] <> EOS) DO (* 字符串比较的循环 *)
            INC(SearchAt);
            INC(NewSymbol)
        END; (* 字符串比较的循环 *)
        IF (StringTable[SearchAt] = StringTable[NewSymbol]) THEN
            NewSymbol := StringStart          (* 找到字符串 *)
        ELSE
            NewSymbol := TableEnd;            (* 不是这一字符串，前进到下一个 *)
        WHILE StringTable[SearchAt] <> EOS DO
```

```
        INC(SearchAt)
    END;
    INC(SearchAt)
END (* 判断是否找到字符串的 IF *)
END; (* 字符串查找的循环 *)
IF NewSymbol = TableEnd THEN (* 将新字符串添加到字符串表中 *)
    INC(NextStringTableEntry);
    TableEnd := NextStringTableEntry
ELSE (* 丢弃新读入的字符串 *)
    NextStringTableEntry := TableEnd
END; (* 添加新字符串的 IF *)
RETURN NewSymbol
END UniqueIdent;
```

线性查找算法不是特别快。由于在一个程序中大量的单词属于标识符，我们有充足的理由寻找一些更快的算法。

3.12.2 基于散列表的实现

符号表快速查找的常见算法是“散列编码”。字符串表中的字符串被划分为 n (n 是某一个较大的数目) 个独立的链表，如图 3-17 所示。所选的数目通常应使得中等大小的被编译程序可将少量标识符（典型情况是平均数接近 1）放到每一个链表（称为散列桶）中，链表中的每一元素指向字符串表中的一个字符串。

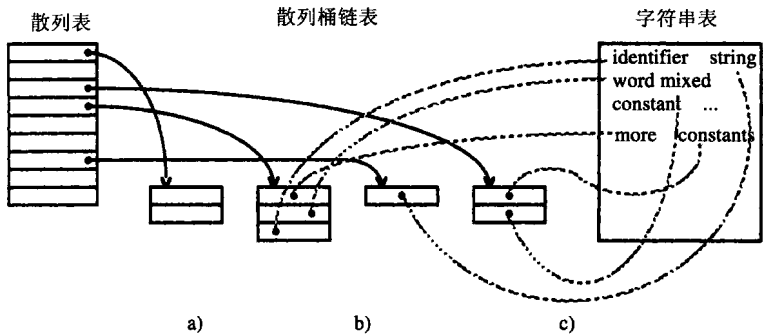


图 3-17 散列表结构。散列表（图 a）是一个指针数组，由字符串表（图 c）中各个字符串的散列函数值作为下标。散列表的每一入口指向一个散列桶（图 b），它是一个由字符串表的下标值组成的链表；一个散列桶包含了那些散列值为散列表中同一下标的所有字符串的引用

散列函数是一个被设计为根据某些数据（通常是组成一个字符串的字符）计算得到一个小数字（称为散列码）的算法，使得散列函数作用于平均混合的字符串时，各个散列码会均匀分布在 $0 \sim n - 1$ 。两个或多个字符串散列到同一下标（从而放入同一桶中）的情况称为碰撞；散列表中出现多个碰撞则会降低其效率。当表中的字符串比桶的数目更多时，这是无法避免的，但将字符串均匀分布到桶中将使得查找时间相对较少。

一个典型散列函数的组成是将所有字符的序数之和以 n 取模；或将字符之间的中间和逐位轮转，以尽量减少相似符号的冲突。编译程序设计人员应记住这样一种常见的编程风格，即在每一相关标识符的前面使用相同的字符串，这会搞糟那些试图节省计算时间（例如仅检查一个标识符的前 3 个字符）的字符串表的散列函数。将序数之和取模的优点是只须在最后有一次除

法就足够了。将正在处理的中间和再次累加到新字符的值会影响到中间和的逐位轮转，仅当中间和达到相当高的阈值时，才有必要用除法（通常较慢）让它回到指定范围。

不同于线性查找算法须检查整个字符串表以判断每一新扫描到的标识符是否已存在，散列表仅须查找单个散列桶中的字符串。如果一个散列表的装载系数低于 50%（即字符串的总数仅有表中散列桶数目的一半），查找时间会有效地降低为常数。每一标识符需额外的时间以计算其散列函数，因而这会其效率。当扫描程序的状态处于扫描一个标识符的下一字符，并将该字符累加到散列码时，其代码可能如下所示：

```
theInputChar := nextInput();
CASE SemanticAction[CurrentState, theInputChar] OF
...
DoIdentChar:
  HashCode := HashCode + HashCode + ORD(theInputChar);
  IF HashCode > 16000 THEN
    HashCode := HashCode MOD HashTableSize
  END;
  StringTable[NextStringTableEntry] := theInputChar;
  INC(NextStringTableEntry);
...

```

若每一散列桶中仅有少量字符串需查找，则传统的字符串比较算法足以胜任。散列桶采用链表会比较浪费内存空间（每一标识符可能花费了加倍的空间），改用一个小的整数数组构造散列桶，可使这一问题得到显著改进。该数组的大小只要比散列桶的平均大小再多一、两个元素即可，若有溢出需处理则将这些数组链接成链表。代码清单 3.6 并未给出这一扩展。

代码清单 3.6 以散列方式实现的字符串表

```
(* 过程 UniqueIdent 用到的全局声明 *)
CONST
  EOS = 0C; (* 字符串结束代码 *)
  HashTableSize = 256; (* 每一字符串以 EOS 为终结 *)
  (* 取决于可用内存、程序平均大小 *)
TYPE
  HashBucket = POINTER TO BucketElement; (* 元素的链表 *)
  BucketElement = RECORD
    theString: INTEGER; (* 字符串表的下标 *)
    nextElement: HashBucket (* 指向散列桶中下一元素的链接 *)
  END;
VAR
  StringTable: ARRAY [1 .. MaxStrings] OF CHAR;
  TableEnd: INTEGER; (* 新符号的起点、旧表的结尾 *)
  NextStringTableEntry: INTEGER; (* 正在处理的新符号的结尾 *)
  HashCode: INTEGER; (* 新符号计算得到的散列码 *)
  HashTable: ARRAY [0 .. HashTableSize] OF HashBucket;
  (* 未使用的入口标记为 NIL *)

PROCEDURE UniqueIdent(): INTEGER;
VAR
  SearchAt, NewSymbol: INTEGER;
  thisBucket: HashBucket;
BEGIN
  NewSymbol := TableEnd; (* 新字符串已保存在结尾 *)

```

```

StringTable[NextStringTableEntry] := EOS;      (* 新字符串的终结 *)
HashCode := HashCode MOD HashTableSize;
thisBucket := HashTable[HashCode];
IF thisBucket = NIL THEN                        (* 若不存在则创建一个新的桶 *)
    NEW(thisBucket);
    HashTable[HashCode] := thisBucket;
    WITH thisBucket^ DO
        theString := NewSymbol;
        nextElement := NIL
    END
ELSE
    WHILE thisBucket <> NIL DO
        WITH thisBucket^ DO
            SearchAt := theString;
            WHILE (StringTable[SearchAt] = StringTable[NewSymbol])
            AND (StringTable[SearchAt] <> EOS) DO (* 字符串比较的循环 *)
                INC(SearchAt);
                INC(NewSymbol)
            END; (* 字符串比较的循环 *)
            IF StringTable[SearchAt] = StringTable[NewSymbol] THEN
                NewSymbol := theString;      (* 找到字符串 *)
                thisBucket := NIL
            ELSIF nextElement = NIL THEN      (* 表中没有, 将它添加该散列桶中 *)
                NewSymbol := TableEnd;
                NEW(nextElement);
                WITH nextElement^ DO
                    theString := NewSymbol;
                    nextElement := NIL;
                    thisBucket := NIL
                END (* WITH nextElement *)
            ELSE                             (* 不是这一字符串, 前进到下一个 *)
                NewSymbol := TableEnd;
                thisBucket := nextElement
            END (* 查找串的 IF *)
        END (* thisBucket 的 WITH *)
    END (* 字符串查找的循环 *)
END;
IF NewSymbol = TableEnd THEN                  (* 将新字符串添加到字符串表中 *)
    INC(NextStringTableEntry);
    TableEnd := NextStringTableEntry
ELSE
    NextStringTableEntry := TableEnd          (* 丢弃新读入的字符串 *)
END; (* 添加新字符串的 IF *)
RETURN NewSymbol
END UniqueIdent;

```

3.12.3 基于查找树的实现

构建并保存一棵单词查找树, 使得任一字符串中的每一字符对应树中的一个结点, 这显著增加了复杂性与表占用的空间, 但性能可略有改进。这种数据结构称为 *trie* (如同 *retrieval* 中那样, 读作 *tree*, 可能是试图命名为一个双关语)。每一结点包含 37 个 (像 *Modula-2* 这类大小写敏感的语言则为 63 个) 链接指向下一比较结点; 标识符中每一个可能出现的字母或数字、再加上一个终结符均有一个链接。然后字符串的比较可与字符的读入同时进行。

扫描一个标识符中的每一字符时，这些字符将用于检索当前的树结点。如果检索到的元素指向下一树结点，则该结点成为下一字符的当前结点。如果检索到的结点元素指向字符串表中的一个字符串，到目前为止它是惟一的；新标识符的后续字符将与字符串表中所选的标识符进行比较。如果它们不匹配，则查找树扩展新的分枝，并且新标识符继续添加到字符串表的结尾。在读入标识符的最后一个字符之后，比较过程也同时结束；如果是一个新标识符，则它已添加到字符串表中。

图 3-18 展示了这种树的结构。由于之前提及的常见标识符命名习惯，将树限定在前几个字符通常是不实用的（尽管这在该查找算法的非编译程序应用中是常见的做法），因为在这种情况下会有大量标识符产生碰撞。代码清单 3.7 假设输入字符已转换为字符编码，其中字母与数字是相邻的。对标准 ASCII 而言，还需要额外的转换（表查找）步骤，以减少结点索引的范围。

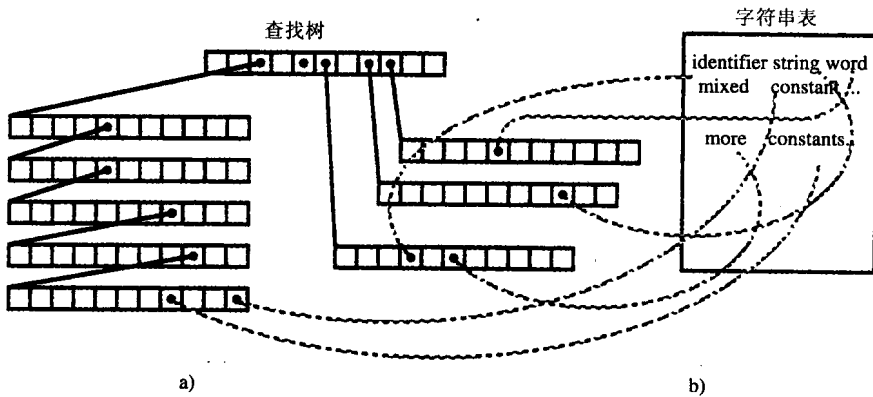


图 3-18 基于树查找的字符串表结构。沿查找树（图 a）的每一条路径定义了一个字符串表（图 b）中一个标识符的拼写，并保持其惟一性。任一树结点中的每一字符下标要么指向另一结点，要么指向一个字符串

代码清单 3.7 以查找树实现的字符串表

```
(* 树查找用到的声明 *)
CONST
  EOS = ';'; (* 字符串结束代码 *)
TYPE
  TreePointer = POINTER TO TreeNode;
  TreeNode = ARRAY ['0' .. 'z'] OF RECORD
    CASE UniqueHere: BOOLEAN OF
      FALSE:
        NextNode: TreePointer | (* 链接到树中下一结点 *)
      TRUE:
        theString: INTEGER (* 指向字符串表的索引 *)
    END (* 不同的 CASE 情况 *)
  END; (* TreeNode *)
VAR
  theInputChar: CHAR; (* 每一字符串以 EOS 结尾 *)
  StringTable: ARRAY [1 .. MaxStrings] OF CHAR;
  TableEnd: INTEGER; (* 新符号的起点、旧表的结尾 *)
  NextStringTableEntry: INTEGER; (* 正在处理的新符号的结尾 *)
  UniqueIdent: INTEGER; (* 返回的字符串 ID *)
```

```

SearchTree, CurrentNode: TreePointer;

matchedString, treeDepth: INTEGER;
newNode: TreePointer;
charindex: CHAR;

BEGIN
    (* 起始代码参阅以下 EndIdentifier *)
    theInputChar := nextInput();
    CASE SemanticAction[CurrentState, theInputChar] OF
        ...
        DoIdentChar:
            StringTable[NextStringTableEntry] := theInputChar;
            IF matchedString = 0 THEN
                WITH CurrentNode^[theInputChar] DO
                    IF UniqueHere THEN
                        matchedString := theString
                        (* 到目前为止是匹配的 *)
                    ELSIF NextNode = NIL THEN
                        matchedString := TableEnd;
                        UniqueHere := TRUE;
                        theString := matchedString;
                        CurrentNode := NIL
                        (* 从此以后是新的字符串 *)
                    ELSE
                        CurrentNode := NextNode;
                        INC(treeDepth)
                        (* 仍沿树中匹配 *)
                    END (* IF UniqueHere 或 NextNode = NIL *)
                END (* WITH CurrentNode^[...] *)
            ELSIF CurrentNode <> NIL THEN
                (* 到目前为止匹配旧的字符串 *)
                IF StringTable[NextStringTableEntry - TableEnd
                    + matchedString] <> theInputChar THEN
                    (* 但不再匹配, 此时创建新的分枝 *)
                    WHILE treeDepth < NextStringTableEntry - TableEnd DO
                        WITH CurrentNode^[StringTable[matchedString + treeDepth]] DO
                            INC(treeDepth);
                            UniqueHere := FALSE;
                            NEW(NextNode);
                            FOR charindex := '0' TO 'z' DO
                                WITH NextNode^[charindex] DO
                                    UniqueHere := FALSE;
                                    NextNode := NIL
                                END (* WITH NextNode^[...] *)
                            END; (* FOR *)
                            CurrentNode := NextNode
                        END (* WITH CurrentNode^[...] *)
                    END; (* WHILE treeDepth *)
                    WITH CurrentNode^[StringTable[matchedString + treeDepth]] DO
                        UniqueHere := TRUE;
                        theString := matchedString
                    END; (* WITH CurrentNode^[...] *)
                    WITH CurrentNode^[theInputChar] DO
                        matchedString := TableEnd;
                        UniqueHere := TRUE;
                        theString := matchedString;
                        CurrentNode := NIL
                    END
                END
            END
    END

```



```
        END (* WITH CurrentNode^[theInputChar] *)
    END (* 不再匹配的 IF *)
END; (* 匹配最早的 IF *)
INC (NextStringTableEntry) |

EndIdentifier:
IF matchedString = 0 THEN
    WITH CurrentNode^[EOS] DO
        IF UniqueHere THEN (* 与一个已有的字符串匹配 *)
            machtedString := theString
        ELSE (* 是一个新字符串, 故添加它 *)
            matchedString := TableEnd;
            UniqueHere := TRUE;
            theString := matchedString
        END (* IF UniqueHere *)
    END (* WITH *)
END; (* IF matchedString = 0 *)
IF matchedString = TableEnd THEN (* 扩展字符串表, 以包含新的串 *)
    StringTable[NextStringTableEntry] := EOS;
    INC (NextStringTableEntry);
    TableEnd := NextStringTableEntry
ELSE (* 丢弃新读入的字符串 *)
    NextStringTableEntry := TableEnd
END; (* IF matchedString = TableEnd *)
UniqueIdent := matchedString; (* 为下一查找而初始化 *)
treeDepth := 1;
CurrentNode := SearchTree |

...

```

3.12.4 不同实现的性能比较

表 3-7 给出了三种字符串表查找算法的相对大小与速度比较。

表 3-7 不同字符串表算法的相对性能

算 法	表的大小 (字节) ①		扫描已有的 12 字节标识符的时间 ②	
	50 个标识符	500 个标识符	50 个标识符	500 个标识符
线性查找	600	6000	2960	29 060
散列表 ③	1400	10 400	250	540
树查找	5720	36 000	250	250

① 假设平均字符串大小为 12 字节, 包括终结符。
② 指虚构的寄存器机器的指令时间, 假设文件存放在内存中。
③ 假设散列表有 100 个字, 每字有 4 个字节。

将整个源程序文件放在内存中还可进一步提高性能; 此时不必建立单独的字符串表, 下标直接指向源代码的数组即可, 从而不再需要字符移动以及逐一读入字符的系统调用。在许多计算机上, 文件可作为一个块直接快速地读入到一个内存数据结构中, 文件将在该数据结构中被使用; 这一读入过程无需 CPU 的参与。

3.13 保留字

大多数程序设计语言定义了一个保留字的集合, 有时也称这些保留字为“字符号”。

Modula-2 语言中的例子是 **PROCEDURE**、**IF**、**THEN**、**END** 等，但 **INTEGER** 或 **TRUE** 却不是（它们只是程序中可重新声明的标识符而已）。保留字作为一个字符，是扫描程序识别并报告给分析程序的独特单词；由于它们具有与标识符相同的形式，保留字成为基于确定 FSA 的扫描程序设计与实现中的一个特殊问题。

略加留意即可得到同时识别标识符单词和特定保留字单词的一个 FSA 的直接实现。识别到一个保留字时，语义动作同时报告保留字单词和通用的标识符单词，实现时必须保证保留字单词的值更优先。然而，单词语言中大量保留字的存在容易导致扫描程序的状态激增。

扫描程序的实现亦可不必显式地识别保留字。在扫描程序的后端，如果识别出一个单词是标识符，则再多用一步测试判断它是否一个保留字。如果在扫描程序的前端已预先将所有保留字装入字符串表中，这一步测试就无需费时的表查找过程，同时也可从字符串表的下标恢复单词的值。单词的值可用字符串表本身的字符进行编码，只要不引起标识符匹配错误即可：将单词的值映射到非字母或数字的字符，这就不成问题了。因而，在扫描程序最后添加的代码只需以下几行：

```
IF NextToken = IDtoken THEN
    NextToken := Token(StringTable[UniqueIdent - 2])
END
```

其中，**IDtoken** 是类型 **Token** 的一个常量，变量 **NextToken**、**StringTable** 和 **UniqueIdent** 的声明如代码清单 3.7 所示。

3.14 使用扫描程序生成工具

现代编译程序很少是以手工方式编写的。分析程序生成工具用于根据一个上下文无关文法构造一个分析程序，并且每一种分析程序生成工具都有一个配套的扫描程序生成工具。

本书重点关注 **TAG** 编译程序，它不仅可构建一个编译程序中的扫描程序和分析程序，而且还可包括约束程序和代码生成程序（**TAG** 是变换属性文法的缩写，本书稍后将讨论；**TAG** 编译程序的更完整描述请参阅附录 B）。**TAG** 编译程序根据两部分构造一个扫描程序。第一部分是扫描程序文法，这一部分采用正则表达式描述各种带名字的单词；在文法的合适地方可加入语义动作的编码。扫描程序的余下部分可从分析程序的文法隐式地推导出来。在一个分析程序的文法中，大多数单词可用加引号的文字量表达清楚，**TAG** 编译程序将这些单词抽取出来，并将它们添加到其扫描程序的构造中。

小结

本章讨论的内容全部围绕着如何开发一个扫描程序。扫描程序是某一正则文法或正则表达式基于对应的有穷状态自动机（FSA）的一个实现。通过构造性方法，我们展示了每一正则表达式和每一正则文法都有一个对应的 FSA；同时用构造性方法说明了每一个由 FSA 接受（或识别）的串的集合，均可表示为一个正则文法或正则表达式。实际上，FSA 与正则表达式之间存在一种非常密切的关系。

正则表达式是表示一个字母表上某一类字符串的一种简明表示法，通常可满足大多数程序设计语言的单词需求。并运算、连接运算以及克林星号运算给正则表达式带来丰富的变化。

本章探讨了两种技术，将一个 FSA 规格说明机械地转换为实现编译程序中的扫描程序的程序设计语言代码。

最后，扫描程序借助于字符串表存储标识符和字符串常量的拼写。字符串表可实现为一个大型的压缩字符数组。本章还讨论了更有效地维护字符串表的几种方法。

符号

- $|$ 正则表达式的并（选择）运算符，读作“或”。例如，正则表达式 R 和 S 的并运算记为 $R|S$ 。
 $*$ 正则表达式的闭包运算符（称为克林星号），读作“任意数目的”或“0 个或多个”。
 L^* 读作“将语言 L 自身连接任意次”。
 L^i 读作“将语言 L 自身连接 i 次”。
 $+$ “1 个或多个”。
 $?$ “0 个或 1 个”。
 Δ 变迁规则的有穷集。
 Q 状态的有穷集。
 Σ 文法或自动机的字母表。

缩略词

- FSA** **Finite-State Automation**, 有穷状态自动机, 正则语言的识别器。
FSM **Finite-State Machine**, 有穷状态机, 等同于 **FSA**。
NDFA **Non-Deterministic Finite Automation**, 不确定的有穷自动机。

关键术语

absorption (吸收律) 对每一正则表达式 R 均满足 $R|R=R$ 的代数定律（亦称正则表达式的幂等性质）。

accept (接受)

- (1) 称语言 L 被一个有穷自动机 M 接受, 当且仅当 L 中每一个串 x 都被 M 接受。
- (2) 串 x 被一个有穷自动机接受, 如果存在一个变迁序列从起始状态出发、且在一个终结状态结束, 使得变迁读入的每一字符与 x 中对应的字符匹配。

alphabet (字母表) 符号的集合 Σ , 其中的符号组成了语言中的句子。

alternation (选择运算) 对正则表达式 R 和 S , $R|S = \{x: x \in R, \text{ 或 } x \in S\}$ 。

distributivity (分配律) 对任意正则表达式 R 、 S 和 T 满足 $R(S|T) = RS|RT$ 的代数定律（即连接运算对选择运算的左分配）。连接运算符也对选择运算右分配。

empty transition (空变迁) 在 **NDFA** 中不必读进任何输入符号即可从一个状态转入另一状态的变迁。

equivalence class (等价类) 是一个有穷状态自动机 M 中的状态的集合 Q_e , 满足 M 对 Q_e 中任意两个状态表现出相同的行为, 即相对于任意的部分串 x , 如果 M 从 Q_e 中的 q 识别 x , 那么它也将从 Q_e 中的 q' 识别 x 。

finite automaton (有穷自动机)

- (1) 状态的有穷集和变迁的有穷集, 变迁基于一个字母表中的输入符号从一个状态转入下一状态。
- (2) 五元组 $(Q, \Sigma, \delta, q_0, F)$, 其中 Q 是状态的有穷集, Σ 是有穷的字母表, δ 是映射 $Q \times \Sigma \rightarrow Q$, q_0 是起始状态, F 是终结状态的有穷集。

halting state (停机状态)

- (1) 一个有穷自动机到达的状态, 此时自动机已处理了所接受输入串的最后一个字符。
- (2) 在有穷自动机 $(Q, \Sigma, \delta, q_0, F)$ 中既属于 Q 也属于 F 的任意 q 。

hash bucket (散列桶) 由一个散列码标识的一个链表。

hash code (散列码) 是一个散列函数的输出。

hash collision (散列碰撞) 指一个散列函数为两个不同的输入字符串计算出相同的散列码。

hash function (散列函数) 是从一个输入字符串到一个整数（典型情况是一个小的正整数）的映射（通常是伪随机的）。

iteration (迭代运算) 正则表达式中一个符号重复出现。

regular expression (正则表达式) 是一种表达一个正则语言中的字符串的表示法, 用迭代、选择、连接、圆括号等运算组织起来。

regular language (正则语言)

- (1) 一个可由一个正则表达式描述的语言。
- (2) 一个可由一个正则文法生成的语言。
- (3) 一个可由一个有穷状态自动机接受的串的集合。

regular set (正则集) 即正则语言。

scanner grammar (扫描程序文法) 是一个正则文法, 定义了组成一个外部(文本)字母表上的所有串的集合, 这些串形成了待编译语言中的单词。

semantic action (语义动作) 是一条规则, 指定了在识别出一个串时所执行的副作用。例如, 将字符串添加到字符串表中, 或定义一个输出单词。

string table (字符串表) 是扫描程序使用的一种数据结构, 用于保存标识符和字符串常量的拼写。

练习

1. 写出以下语言的正则表达式。

- (a) 由英文字母组成的所有串, 其中每个串依次包含了 5 个元音字母, 且每个元音字母在整个串中仅出现 1 次。
- (b) 由英文字母组成的所有串, 其中每个串由依次包含了 5 个元音字母的子串组成, 且每个元音字母在整个串中可出现多次; 也允许每个元音字母在其位置连续出现多次, 例如 *bghaaatrasdfewqprilkohmoooozu* 是一个可接受的子串(因而也是一个可接受的串)。
- (c) 由 0 和 1 组成的所有串, 其中每个串有偶数个 0 和奇数个 1。
- (d) 由 0 和 1 组成的所有串, 其中每个串均不包含子串 011。
- (e) 由 1、2 和 3 组成的所有串, 其中每个串中 1、2 或 3 出现不超过 1 次。
- (f) 由 1 和 2 组成的所有串, 其中每个串中 1 不会重复(即没有子串 11), 且至少有一个 1 或 2; 不允许有空串。
- (g) 由 1 和 2 组成的所有串, 其中每个串中不会有重复的数字(即没有子串 11 和 22), 且至少有一个 1 或 2; 不允许有空串。
- (h) 由 a 和 b 组成的所有串, 其中每个串的长度为 5 个或更少的字符; 允许有空串。

2. 描述以下正则表达式所表达的语言。试用自然语言的描述捕捉这些语言的本质特性, 而不要简单地将正则表达式符号转为自然语言表达。

- (a) $0(011)^*0$
- (b) $((\epsilon \mid 0)1^*)^*$
- (c) $(011)^*0(011)(011)$
- (d) $0^*10^*10^*10^*$
- (e) $(00111)^*((01110)(00111)^*(01110)(00111)^*)^*$
- (f) $(00111)^*(01110)((01110)(00111)^*(01110)(00111)^*)^*((01110)(00111)^*110)1(00111)^*1$

3. 改写以下正则表达式, 消除其中的+和?运算符。

- (a) $01(01?110)^+$
- (b) $((11(010)^+)?)(1101(10)?)^+$

4. 通过连续应用代数定律, 化简以下正则表达式。

- (a) $0121321(10101)1013$

(b) $(12|13)(0|1)|((2|3)|((3|1)(1|0)))$

5. 指出以下 FSA 哪些是等价的, 哪些是同构的。

(a)

	0	1
S	A	B
A	S	C
*B	C	S
C	B	A

(b)

	0	1
S	B	A
*A	C	D
B	D	C
C	A	B
D	B	A

(c)

	a	b
E	F	G
F	E	H
*G	H	E
H	G	F

(d)

	0	1
J	M	K
*K	L	N
L	K	M
M	N	L
N	M	K

6. 将以下正则文法转换为正则表达式。

(a) $A \rightarrow aB$

$A \rightarrow bA$

$B \rightarrow aC$

$B \rightarrow cD$

$C \rightarrow c$

$D \rightarrow d$

(b) $A \rightarrow 0B$

$A \rightarrow 1C$

$B \rightarrow 0C$

$B \rightarrow 1C$

$C \rightarrow 0D$

$C \rightarrow 1D$

$D \rightarrow 0A$

$D \rightarrow 1A$

$D \rightarrow 0$

(c) $A \rightarrow 1B$

$A \rightarrow 0C$

$B \rightarrow 0C$

$B \rightarrow 1B$

$C \rightarrow 1B$

$C \rightarrow 1$

(d) $S \rightarrow 0A$

$S \rightarrow 1B$

$S \rightarrow 1$

$A \rightarrow 0S$

$A \rightarrow 1C$

$B \rightarrow 0C$

$B \rightarrow 1S$

$C \rightarrow 1A$

$C \rightarrow 0B$

$C \rightarrow 0$

7. (a) 将练习 6 中的正则文法转换为不确定的有穷状态自动机。

(b) 将练习 7 (a) 中的 NDFA 转换为约简的确定 FSA。

8. 将以下正则表达式转换为正则文法。

(a) $(12|2)^*(12|1|2)$

(b) $1^*(0|01)^*$

(c) $a^*|(bca^*(ab|ba))$

(d) $((0|1)^*(10|01))^*1|(11|00)$

9. (a) 将练习 8 中的正则表达式转换为不确定的有穷状态自动机。

(b) 将练习 9 (a) 中的 NDFA 转换为约简的确定 FSA。

10. 将以下不确定的有穷状态自动机转换为确定的有穷状态自动机。

(a)

	a	b	c	ϵ
S	A,B	C,D	D	A,B,D
A	A		C	B
B	A	D		C
*C	B	A		A
*D	C	B		S

(b)

	a	b	c	ϵ
S	A	B	C	
*A	S	B,C		C,D
B	A	B	A,D	A
*C	A,B	B		A,D
D		A,C	B	S

(c)

	a	b	c	ϵ
S	C,D	A	D	
A	A	B	C	D
*B		C	D	C
C	A,B		C	
D	C	B		A

11. (a) 根据正则表达式 $(a|b)^*aaba$ 构造一个确定的有穷状态自动机。

(b) 编写一个扫描程序实现练习 11 (a) 得到的 FSA。

12. 根据以下每一 FSA 分别构造等价的正则文法。

(a)

	a	b	c
S	A	B	
A	A		C
*B		C	D
C	B		C
D	C	B	

(b)

	a	b	c
S	A	A	D
A	A	B	C
B		C	D
C	B		C
*D	C	B	

13. 给出可由一个有穷状态自动机 M 接受的语言 L 的形式化定义。

14. 给出图 3-16 所示 FSA 不接受的一个语言实例 L, 再指定 L 中的一个串 x, 使得 $\delta(q_0, x) \neq p$, 其中 p 不属于图 3-16 所示的终结 (停机) 状态集 F。给出 L 的文法。

15. 给出图 3-16 所示 FSA 不接受的一个语言实例 L' , 但其中至少有一个串 x 可被同一 FSA 接受。给出 L' 的文法。
16. 给出练习 1 中每一正则表达式的正则文法。
17. 给出练习 1 中每一正则语言的 FSA。
18. 证明由一个 FSA 识别的语言可由一个正则表达式表示。
19. 证明对任一正则表达式 R , 存在一个有穷自动机 M 接受该正则表达式 R 表示的串的集合。
20. (引理 3.1) 证明每一个包含单个串的集合都有一个对应的正则表达式。提示: 使用基于构造的证明技术。
21. (命题 3.1) 证明或反驳每一正则语言都是有穷的这一命题。
22. (定理 3.2) 证明如果 L_1 和 L_2 是正则语言, 则 $L_1 \mid L_2$ 、 $L_1 \bullet L_2$ 和 L_1^* 分别也是正则语言。
23. (引理 3.2) 证明如果 L 是一个正则语言, 则对任意 $n \geq 0$, L^n 也是正则语言。
24. (定理 3.3) 如果 L 是一个正则语言, 则下列语言也是正则语言。
 - (a) $L^n \mid L^m$, 其中 $n, m \geq 0$
 - (b) $L^1 = \{\epsilon\} L$
25. 证明对任一正则表达式 R , R^* 等于 $(R^*)^*$ 。
26. 证明表 3-2 中的每一恒等式。
27. 对正则表达式 R 和 S , 给出一个 RS 不等于 SR 的例子, 从而说明 RS 并不总是等于 SR 。
28. 半群是一个二元组 (T, \oplus) , 其中 T 是元素的非空集, \oplus 是一个定义在 T 上的、具有结合性的二元运算。给出本章定义的两个半群的例子。
29. 证明如果 R 是一个正则集, 则存在一个右线性文法 G , 使得 G 定义的语言 $L(G)$ 是 R 。

复习小测验

指出下列陈述是否正确。假设 R 、 S 和 T 分别代表 Σ^* 中句子的正则集, $\{\epsilon\}$ 是一个仅含空串 ϵ 的集合。

1. 空语言是一个正则表达式。
2. 如果 $R = S^* T$ 则 $R = SR \mid T$ 。
3. $(R^* S)^* = (R \mid S)^* S \mid \{\epsilon\}$ 。
4. 每一有穷字符串均可由多于一个的正则表达式表示。
5. 每一正则语言均包含 $\{\epsilon\}$ 作为子集。
6. 每一正则表达式均定义了一个正则语言。
7. 所有带克林星号的正则表达式都隐式地使用了连接运算。
8. $(R \{\epsilon\} S \{\epsilon\} T)^* = (RST)^* (RST)^*$ 。
9. $\{\} \{\epsilon\} = \{\}$ 。

编译程序实验项目

1. 编写一个正则表达式, 生成由 Itty Bitty Modula 语言的全部单词组成的语言, 这些单词要么是由你在第 2 章所编写的文法所定义的, 要么是附录 A 语法图的椭圆中所示的全部单词。正则表达式生成的每一句子, 应恰好由一个单词组成。请务必将空格和注释考虑在内, 并给出它们的合适语义。
2. 机械地构造 (即不要重新考虑该问题) 一个有穷状态自动机识别 Itty Bitty Modula 语言的单词, 可从以下两种途径构造:
 - (a) 你编写的正则表达式 (如上述第 1 题)。
 - (b) 你为同一语言编写的正则文法。
3. 编写一个扫描程序的过程, 恰好实现上述第 2 题中的 FSA。使用以下主程序测试你的扫描程序 (若采用 Pascal 语言或其他语言编写, 则使用其等价形式):

```
MODULE MainProgram;
...
VAR
```

(* 你负责填写其他类型和变量 *)

```

NextToken: Token;
...
BEGIN
  InitializeScanner;
  REPEAT
    Getoken;
    WriteInt (ORD(NexToken));
    WriteLn
  UNTIL NextToken = Dot;
END MainProgram.

```

(* 你负责填写 Getoken 和其他过程 *)

进一步阅读

- Chomsky, N. "On Certain Formal Properties of Grammars." *Information and Control*, Vol.2 (1959), pp.137-167.
参阅第 5 节, 其中讨论了正则文法。
- Clifford, A.H. & Preston, G.B. *The Algebraic Theory of Semigroups*. Providence, RI: American Mathematical Society, 1961.
参阅第 1.1 节与半群相关的基本定义。
- Cohen, D.I.A. *Introduction to Computer Theory*. New York: Wiley, 1986.
参阅第 7 章关于克林定理的介绍, 以及第 10 章关于正则语言的介绍。
- DeRemer, F.L. "Lexical Analysis." In *Compiler Construction* ed. F.L. Bauer & J. Eikel. New York: Springer-Verlag, 1974.
参阅第 3.1.3 节关于正则文法转换为正则表达式的介绍。
- Ginzburg, A. *Algebraic Theory of Automata*. New York: Academic Press, 1968.
参阅第 4 章关于正则表达式的介绍, 特别是第 4.1~4.6 小节给出的克林定理证明。
- Gries, D. *Compiler Construction for Digital Computers*. New York: Wiley, 1971.
参阅第 9 章关于字符串表 (D. Gries 称之为符号表) 组织方法的详尽讨论。
- Hopcroft, J.E. & Ullman, J.D. *Introduction to Automata Theory, Languages, and Computation*. Reading, MA: Addison-Wesley, 1979.
参阅第 2.5 节关于正则表达式的介绍。
- Kleene, S.C. "Representation of Events in Nerve Nets and Finite Automata." In *Automata Studies*. Princeton, NJ: Princeton University Press, 1956, pp.3-42.
- Knuth, D.E. "On the Translation of Languages from Left to Right." *Information and Control*, Vol.8, No.6 (1965), pp.607-639.
参阅第 611 页关于正则语言 (有穷自动机) 的介绍; 本文给出了许多有趣的文法。
- McKeeman, W.M. "Symbol Table Access." In *Compiler Construction* ed. F.L. Bauer & J. Eikel. New York: Springer-Verlag, 1974.
参阅第 7 节关于散列表访问的介绍, 以及第 8 节关于散列函数的介绍。
- Mossenbock, H. "Alex - A Simple and Efficient Scanner Generator." *ACM SIGPLAN Notices*, Vol.21, No.12 (December 1986), pp.139-148.
参阅第 146~147 页关于扫描程序文法的一个例子。Alex 已采用 Modula-2 语言实现。
- Sebesta, R.W. & Taylor, M.A. "Minimal Perfect Hash Functions for Reserved Word Lists." *ACM SIGPLAN Notices*, Vol.20, No.12 (December 1985), pp.47-53.
参阅第 49 页关于一种新的完美散列函数的介绍, 可将其用于 Modula-2 语言的保留字。
- Shyu, Y.-H. "From Semi-Syntactic Analyzer to a New Compiler Model." *ACM SIGPLAN Notices*, Vol.21, No.12 (December 1986), pp.149-157.
参阅第 152 页关于扫描程序的有穷状态自动机的完整介绍。

第 4 章 分析程序和上下文无关语言

本章旨在：

- 介绍 $LL(k)$ 文法
- 阐明消除左递归和左因子的转换规则，以获得 $LL(k)$ 的上下文无关文法
- 确定上下文无关文法及其对应的下推自动机之间的关系
- 介绍 *First*、*Follow* 和 *Selection* 集，用于确定文法中的一个非终结符是否 $LL(k)$ 的，以及文法本身是否 $LL(k)$ 的
- 为上下文无关文法扩展正则表达式运算符
- 学习如何将一个扩展的 $LL(1)$ 上下文无关文法转换为一个递归下降分析程序

4.1 简介

在编译程序中，扫描程序从输入串的字符序列识别出语言的终结符，这些终结符是编译程序所识别语言的单词。由扫描程序定义的语言是扫描程序所识别的单词的集合，这是程序设计语言词法层面的东西；分析程序则是一个下推自动机（一个栈机器，通常缩写为 PDA），负责识别语言的短语结构，即语言中的单词如何正确地组织在一起，形成一个语法上正确的程序。

大多数程序设计语言允许在句子的构造中包含嵌套且匹配的括号。如第 2 章所述，这类语言是上下文无关的，并可由上下文无关文法（CFG）定义。由 PDA 识别的每一语言均是上下文无关的；这一点的证明相当容易，即根据一个 PDA 的变迁可构造一个 CFG 的规则。为一个 PDA 所接受的语言设计的 CFG 规则，可令 CFG 中的每一产生式对应着 PDA 中的一条变迁。本章还说明了每一上下文无关文法均可被一个不确定的 PDA 接受。

本章的重点是由一类确定的 PDA 所接受的上下文无关语言，这类语言由一种名为 $LL(k)$ 的上下文无关文法定义。一个 $LL(k)$ 文法允许在输入串上向前看 k 个符号即可完成确定的、自顶向下的分析。“LL”这一写法描述了所用的分析策略，即 LL 分析策略从左到右扫描输入串，LL 分析程序构造了一个最左推导。本章介绍一种将 $LL(1)$ 文法转换为一个由传统程序设计语言编写的所谓递归下降分析程序。

4.2 下推自动机

下推自动机的形式化定义是一个七元组：

$$P = (\Sigma, Q, \Delta, H, h_0, q_0, F)$$

其中的所有构成除与 FSA 相同之外，再加上两个新的部分：有穷的栈字母表 H ；属于 H 的初始符号 h_0 ，作为栈的初始内容。类似一个 FSA，当 PDA 到达输入串结尾，并处于某一终结状态（集合 F 中的一个状态）时，停机并接受该输入串；它也可在栈为空时停机并接受输入串。如果 PDA 在其他情况下到达输入串结尾，或 PDA 阻塞，则它拒绝输入串。

栈字母表可以有也可以没有与输入字母表相同的符号（这两种类型我们都将处理到），但栈本身可伸展到任意深度。我们并不称之为无穷栈（因为如此一来，填充它将需要无限长），但其深度没有固定的有限长度限制。

变迁规则集 Δ 是从状态、输入字母表和栈字母表映射到状态和栈字母表上的串的偏函数，

即集合 Δ 中的一条变迁规则 δ 具有如下函数式:

$$\delta: Q \times (\Sigma \cup \epsilon) \times H \rightarrow Q \times H^*$$

这意味着每一变迁是为一个状态定义的, 它要么读入一个输入单词, 要么什么也不读入, 但总是从栈中弹出一个符号, 然后前进到一个新状态, 并将 0 个或多个符号组成的串压回栈中。变迁规则类似于 FSA 的变迁规则:

$$\delta(q_i, \alpha, \eta) = (q_j, x)$$

其中, q_i 和 q_j 是状态, α 是输入字母表中的某一单词或 ϵ , η 是栈字母表中的一个单词, x 是栈字母表上的字符串 (可能为空)。如果变迁不影响栈, 那么串 x 也可直接又是符号 η 。一个格局 (q, ω, x) 表示 PDA 处于状态 q 时, 待输入的未读入符号串是 ω , 并且栈顶有一个特定的串 x 。

与 FSA 一样, 一个 PDA 也可能是不确定的。不确定的 PDA 与确定的 PDA 主要区别在于, 对任一给定格局存在多于一个的可能变迁; 即从同一状态、相同栈符号出发, 有 2 个或更多的变迁规则要么读入同一输入单词, 要么什么都不读入。

例如, 令 $P_0 = (\{a, b, c\}, \{A, B, C\}, \Delta, \{h, i\}, i, A, \{\})$ 是一个确定的 PDA (如图 4-1 所示), 其中 Δ 是如下变迁的集合:

$\delta(A, a, i) = (B, h)$

$\delta(B, a, h) = (B, hh)$

$\delta(C, b, h) = (C, \epsilon)$

$\delta(A, c, i) = (A, \epsilon)$

$\delta(B, c, h) = (C, h)$

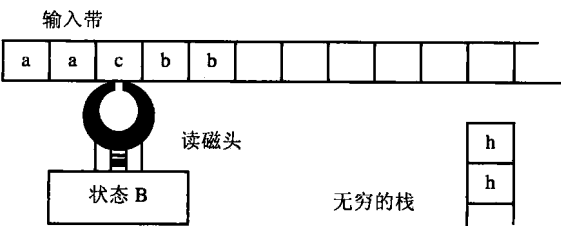


图 4-1 一个下推自动机的例子

该 PDA 识别由文法 $\{ S \rightarrow aSb, S \rightarrow c \}$ 产生的所有串, 即由相等数目的 a 和 b 、中间以单个 c 分隔的串。它的起始格局是状态 A 以及栈中有一个 i , 在栈为空时停机。为它提供输入串 $aacbb$ 时, 它在执行了 5 步变迁后停机, 并接受该串 (注意栈顶从左边开始; 图 4-2 也演示了这一过程):

格局	PDA 动作
$(A, aacbb, i)$	初始格局: 读入 a , 弹出 i , 压入 h , 再转入状态 B
$(B, acbb, h)$	读入 a , 弹出 h , 压入 hh , 转入状态 B
(B, cbb, hh)	读入 c , 弹出 h , 压入 h , 转入状态 C
(C, bb, hh)	读入 b , 弹出 h , 转入状态 C
(C, b, h)	读入 b , 弹出 h , 转入状态 C
(C, ϵ, ϵ)	停机

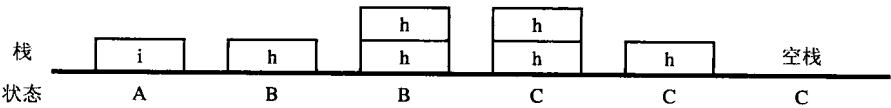


图 4-2 下推自动机的栈的增长



4.2.1 停机条件的等价性

显而易见, 在一个终结状态停机等价于栈为空时停机, 即对每一个在终结状态停机的 PDA, 可构造一个等价的 PDA (接受同一语言) 在栈为空时停机; 反之亦然。令 P_1 为空栈停机的 PDA, 即其终结状态的集合 F 为空。我们构造一个与 P_1 等价的下推自动机 P_2 , 不同之处在于它有 2 个新加的状态 q_f (指派为它的终结状态) 和 q_g (指派为它的起始状态), 还有一个新加的栈符号 h_f (指派为初始栈元素) 和以下新加的变迁规则:

$$\delta(q_g, \varepsilon, h_f) = (q_0, h_0 h_f) \quad (\text{栈顶在左边})$$

$$\delta(q_i, \varepsilon, h_f) = (q_f, h_f) \quad \text{对 } P_1 \text{ 的每一状态 } q_i \in Q_1$$

第一条新规则将原来的初始栈符号 h_0 压入到新的初始栈符号 h_f 之上, 并转入原来的起始状态 q_0 , 无须读进任何输入; 从此 P_2 会像 P_1 那样操作, 直至到达 P_1 中栈已为空的情形; 在 P_2 的栈中仍有 h_f , 由于 P_1 中任一状态从该格局均有变迁转入停机状态 q_f (如上述第二条规则所示), 因而 P_2 恰好在同一格局停机, 只不过多执行了一次变迁。由于新加的变迁全部不读进任何输入, 并且要求见到新的栈符号 h_f 才能继续前进, 所以它们不会影响 P_1 的原有功能。若对 PDA 例子 P_0 应用这一转换, 结果将是:

$$P_0' = (\{a, b, c\}, \{A, B, C, E, F\}, \Delta', \{h, i, f\}, f, E, \{F\})$$

其中, Δ' 是变迁的扩展集合:

$$\delta(A, a, i) = (B, h) \quad \delta(A, c, i) = (A, \varepsilon)$$

$$\delta(B, a, h) = (B, hh) \quad \delta(B, c, h) = (C, h)$$

$$\delta(C, b, h) = (C, \varepsilon)$$

$$\delta(E, \varepsilon, f) = (A, if) \quad \delta(A, \varepsilon, f) = (F, f)$$

$$\delta(B, \varepsilon, f) = (F, f) \quad \delta(C, \varepsilon, f) = (F, f)$$

现在, 给定任一 PDA (设为 P_2) 在一个终结状态停机; 类似地, 我们构造一个如下的新 PDA (设为 P_3) 在栈为空时停机, 如下所示。我们为 P_2 添加一个新状态 q_g , 对 H 中的每一符号 h_i 添加以下变迁:

$$\delta(q_g, \varepsilon, h_i) = (q_g, \varepsilon)$$

$$\delta(q_f, \varepsilon, h_i) = (q_g, \varepsilon) \quad \text{对 } P_2 \text{ 的每一终结状态 } q_f$$

从而只要 P_2 因到达停机状态 q_f 而停机, P_3 将进入特殊状态 q_g 清空栈 (从而停机)。

将上述转换应用到 P_0' , 可得

$$P_0'' = (\{a, b, c\}, \{A, B, C, E, F, G\}, \Delta'', \{h, i, f\}, f, E, \{\})$$

其中, Δ'' 是变迁集 (再次扩展):

$$\delta(A, a, i) = (B, h) \quad \delta(A, c, i) = (A, \varepsilon)$$

$$\delta(B, a, h) = (B, hh) \quad \delta(B, c, h) = (C, h)$$

$$\delta(C, b, h) = (C, \varepsilon)$$

$$\delta(E, \varepsilon, f) = (A, if) \quad \delta(A, \varepsilon, f) = (F, f)$$

$$\delta(B, \varepsilon, f) = (F, f) \quad \delta(C, \varepsilon, f) = (F, f)$$

$$\delta(G, \varepsilon, h) = (G, \varepsilon) \quad \delta(F, \varepsilon, h) = (G, \varepsilon)$$

$$\delta(G, \varepsilon, i) = (G, \varepsilon) \quad \delta(F, \varepsilon, i) = (G, \varepsilon)$$

$$\delta(G, \varepsilon, f) = (G, \varepsilon) \quad \delta(F, \varepsilon, f) = (G, \varepsilon)$$

4.2.2 根据上下文无关文法构造下推自动机

构造出来的 PDA 的输入字母表与文法的字母表相同；它的栈字母表由 $(\Sigma \cup N)$ 组成，即输入字母表中的所有单词加上文法的所有非终结符；初始栈符号是文法的目标符号。该 PDA 仅有一个状态 q ，并在栈为空时停机。它的变迁按如下方式构造：为 Σ 中的每一终结符 x ，添加一条变迁规则：

$$\text{N.1} \quad \delta(q, x, x) = (q, \varepsilon)$$

该变迁读入一个单词，并从栈中弹出相同的单词。对 G 中的每一产生式 $A \rightarrow \omega$ ，添加一条变迁规则：

$$\text{N.2} \quad \delta(q, \varepsilon, A) = (q, \omega)$$

该变迁将栈顶的一个非终结符替换为该产生式右部的串 ω （由终结符和非终结符组成），无须读入任何输入。

显然，该 PDA 实现了一种自顶向下的语法分析。从栈中的目标符号出发，它不确定地连续重写最左的非终结符，直至句型中的最左符号（即栈顶符号）是一个与输入串首个单词匹配的终结符。此时执行读入该单词的变迁，这一过程不断重复，直至已读完整个输入串或 PDA 阻塞。PDA 阻塞当且仅当不存在变迁序列最终使得下一输入单词与栈顶符号匹配，这意味着语言中不存在与输入串匹配的串。注意仅当从输入串中读入首个符号之前，PDA 的栈才完全表示了当前的句型；此后的句型均分为两部分：已读入的输入串部分和当前的栈。刚才才是以构造方式证明了定理 4.1。

定理 4.1 如果 L 是一个上下文无关语言，则存在一个不确定的 PDA 接受该语言。

引理 4.1 如果 L 是被一个 PDA 接受的语言，则 L 是上下文无关的。

【证明】以构造方式证明。简要地说，令 M 是一个 PDA，根据 M 的变迁构造出文法的规则。这些规则的具体构造方法留作练习 17。 \square

定理 4.2 如果 L_1 和 L_2 是上下文无关语言，则其并集 $L_1 + L_2$ 也是上下文无关的。

【证明】以构造方式证明，两次应用定理 4.1 的结论；再从识别 L_1 的 PDA₁ 和识别 L_2 的 PDA₂ 构造出一个新的 PDA；最后使用引理 4.1 的结论（参见练习 18）。 \square

注意，定理 4.2 的证明方式也可通过将 L_1 的 CFG₁ 和 L_2 的 CFG₂ 混合在一起，构造出一个新的上下文无关文法。

定理 4.3 如果 L_1 和 L_2 是上下文无关语言，则 L_1 和 L_2 中所有串的连接（记为 $L_1 \bullet L_2$ ）也是一个上下文无关语言。

【证明】可通过构造方式证明。不失通用性，可假设 L_1 和 L_2 分别由上下文无关文法 CFG₁ 和 CFG₂ 定义，如下所示：

CFG₁ = (Σ, N_1, P_1, A) 定义了语言 L_1

CFG₂ = (Σ, N_2, P_2, A) 定义了语言 L_2

其中， (Σ, N_1, P_1, A) 和 (Σ, N_2, P_2, A) 分别是 CFG₁ 和 CFG₂ 的字母表、非终结符集、产生式集和目标符号。我们有意将这两个文法的目标符号命名为同一名字 A 。由于需要区别这两个文法的非终结符，我们将 CFG₁ 的所有非终结符加上下标 1，将 CFG₂ 的所有非终结符加上下标 2。现在这两个文法具有如下形式：

CFG₁ = (Σ, N, P, A_1) 定义了语言 L_1

$CFG_2 = (\Sigma, N, P, A_2)$ 定义了语言 L_2

然后引入一个新的非终结符 A 和以下重写规则：

$A \rightarrow A_1 A_2$

其中， A_1 是 CFG_1 的目标符号， A_2 是 CFG_2 的目标符号，从而 $L_1 \bullet L_2$ 中的任意串都可由这个新文法（称之为 CFG_A ）推导出。如果 ω 是 CFG_A 推导出的一个串， ω 将具有如下形式：

$\omega = \{ L_1 \text{ 中的串 } \} \{ L_2 \text{ 中的串 } \}$

由于原文法中的非终结符可通过其下标加以区分，在推导过程中选择下一重写规则时是不会混淆的。 □

考虑文法 G_{19} ，它产生的串是若干 a 后接着与 a 个数相同的 b ：

P.1 $S \rightarrow a S b$

P.2 $S \rightarrow \epsilon$

新构造的 PDA 变迁如下所示：

4.1 $\delta(q, \epsilon, S) = (q, aSb)$ (根据 N.2 和 P.1)

4.2 $\delta(q, \epsilon, S) = (q, \epsilon)$ (根据 N.2 和 P.2)

4.3 $\delta(q, a, a) = (q, \epsilon)$ (根据 N.1)

4.4 $\delta(q, b, b) = (q, \epsilon)$ (根据 N.1)

分析输入串 $aabb$ 时，该 PDA 从初始格局 $(q, aabb, S)$ 出发，如表 4-1 的第 1 行所示。容易看出，可用的变迁只有规则 4.1 和 4.2；假设我们选择规则 4.1，产生了第 2 行所示的新格局。此时，惟一可能的变迁是规则 4.3，从而产生格局 (q, abb, Sb) 。接着我们又面临同样的选择，假设我们选择规则 4.2，这让我们得到第 4a 行的格局，其中已无变迁可用，因而该选择是错误的；如果我们选择再次应用规则 4.1，它后面还可应用规则 4.3，从而产生第 5 行的格局 (q, bb, Sbb) 。第三次应用规则 4.1 时在格局 $(q, bb, aSbbb)$ 受阻，但规则 4.2 可成功进入第 6b 行的格局，从该格局出发两次应用规则 4.4 就可读入剩余的输入串并将栈清空。

表 4-1 串 $aabb$ 的不确定分析

	输入串（小数点后表示下一读入单词）	栈	变迁规则
1	$.aabb$	S	
2	$.aabb$	aSb	4.1
3	$a.abb$	Sb	4.3
4a	$a.abb$	b	4.2 ?
4b	$a.abb$	$aSbb$	4.1
5	$aa.bb$	Sbb	4.3
6a	$aa.bb$	$aSbbb$	4.1 ?
6b	$aa.bb$	bb	4.2
7	$aab.b$	b	4.4
8	$aabb.$		4.4

如本书第 3 章所述，不确定的自动机不会产生一个非常高效的计算机程序，上述这一 PDA 也不例外。敏锐的读者可能早已注意到，通过观察输入串中的下一单词，并与可用的选项进行比较，可帮助决定一些不确定的变迁规则选择。诚然，这正是 $LL(k)$ 预测分析所采用的做法。

4.3 LL(k)条件

从文法构造的 PDA 有两个变迁导致上一 PDA 是不确定的。这两个变迁都无须读进任何输入，并且都是从栈中弹出非终结符 S ，然后用其他符号取而代之。

其中一个变迁将一个以终结符 a 开头的串压入栈中；在 a 位于栈顶的情况下，如果想要分析程序不阻塞，则下一输入符号必须也是一个 a 。另一个变迁弹出 S ，并在其位置不压入任何符号。除初始格局外，每一个压入栈中的 S 都属于一个在 S 后面跟着终结符 b 的串的一部分；当弹出 S 并用空串取而代之时， b 将暴露在栈顶，因而下一输入符号必须也是 b 才可防止分析程序的 PDA 阻塞。

因而在上述例子中，两个不确定的选择强制规定了下一变迁对不同输入符号的需求。作反向思维，如果我们观察下一输入单词（但并不读入它），则可明智地选择在下一步该输入符号不会引起阻塞的变迁。这意味着如果下一输入单词是 a ，则应选择规则 4.1 的变迁，因为它使得下一步有可能应用规则 4.3 的变迁；而如果下一输入单词是 b ，则应选择规则 4.2，因为它使得下一步有可能应用规则 4.4。仅观察下一输入单词而不读入它，称为“向前看”。如果在输入带上只要向前看 k 个符号，就可从一个文法构造出一个确定的自顶向下 PDA，则称该文法为 LL(k)文法。

文法 G 是 LL(k)的，如果任意两个最左推导：

$$S \Rightarrow^* u A z \Rightarrow u x z \Rightarrow^* u v w$$

$$S \Rightarrow^* u A z \Rightarrow u y z \Rightarrow^* u v w$$

都有 $x=y$ ；其中， u 、 v 和 w 都是由终结符组成的串（也可能是空串）， x 、 y 和 z 都是由终结符和非终结符组成的串，且 $|v|$ （串 v 的长度）是 k 个符号。这意味着不存在两个不同的产生式 $A \rightarrow x$ 和 $A \rightarrow y$ ，使得从应用这两条产生式开始，在生成的串中有同样的 k 个符号。注意，我们并不要求 x 和 y 本身生成有 k 个符号的 v ，而只要求不管它们生成什么串，在后面接上由串 z 生成的终结符凑成 k 个单词后，会产生相同的串。换言之，根据文法 G 构造的一个自顶向下分析程序读入前缀 u 中的单词后，在面临产生式 $A \rightarrow x$ 和 $A \rightarrow y$ 的抉择时，只要该文法是 LL(k)的，就可根据未读入的剩余输入串中的前 k 个符号，确定地选择其中的一条产生式。

回到上述例子，可见该文法是 LL(1)的，因为向前看一个符号即可消除所有的不确定性。文法 G_{19} 的产生式 P.1 中，产生式右部的首个单词（即 a ）是帮助我们基于该产生式选择变迁（即规则 4.1）的向前看符号。产生式 P.2 的右部没有任何单词，即其右部为空。但我们可通过观察其他产生式的右部，发现那些可跟在该产生式左部的非终结符之后的单词。因而文法 G_{19} 中的推导

$$S \Rightarrow^* \underline{a S b} \Rightarrow a a S b b \Rightarrow^* a a b b$$

要求应用产生式 $S \rightarrow aSb$ （下划线所示步骤），对相应的 PDA 而言，分析到这一步时未读入的剩余输入串 abb 中的下一向前看符号（ $k=1$ ）是 a ，这正是所选产生式的首个单词。另一方面，推导

$$S \Rightarrow^* \underline{a S b} \Rightarrow a b \Rightarrow^* a b$$

要求应用产生式 $S \rightarrow \epsilon$ ，因为 PDA 分析到这一步时，输入串的下一符号（也是仅存的符号）是 b ，而在产生式 P.1 中该单词可跟随在 S 之后。

4.3.1 First 和 Follow 集

为证明一个文法是 $LL(k)$ 的, 我们基于文法的产生式构造由长度为 k 的串组成的一些集合: 为文法产生式的所有右部 w 构造 $First_k(w)$; 为文法的所有非终结符 N 构造 $Follow_k(N)$; 再据此可构造所有产生式的选择集, 然后判断该文法的 $LL(k)$ 条件是否成立。

对任意串 w , $First_k(w)$ 是从 w 推导出的、由 k 个或少于 k 个单词组成的所有终结字符串的集合。非终结符 N 的 $Follow_k$ 是由 k 个单词组成的终结字符串的集合, 并且这些终结字符串可跟在由 N 推导出的任何串之后。一条产生式的选择集 $Select_k$ 是由 k 个单词组成的向前看符号串的集合, 这一集合在一个确定的自顶向下分析程序中决定了对产生式的选择。

对任意由终结符和非终结符组成的串 u 、 v 和 w , 可根据表 4-2 中的规则 F.1~F.4 构造集合 $First_k(w)$ 。注意在这些规则中, $First_k$ 既可应用于单个串, 也可应用于串的集合, 这可根据上下文来确定。

表 4-2 First 和 Follow 集的定义

F.1	$First_k(uv) = First_k(First_k(u)First_k(v))$	
F.2	$First_k(N) = \bigcup (First_k(w))$	对所有形如 $N \rightarrow w$ 的产生式中的 w
F.3	$First_k(x) = \{x\}$	对文法字母表中的所有终结符 x
F.4	$First_k(\epsilon) = \{\epsilon\}$	
F.5	$Follow_k(A) = \bigcup (First_k(First_k(v)Follow_k(B)))$	对所有产生式 $B \rightarrow uAv$

规则 F.1 表明, 对于一个由两个子串组成的串, 其 $First_k$ 可通过各个子串的 $First_k$ 来构造, 形成的串由第一个子串的 $First_k$ 中的一个元素连接第二个子串的 $First_k$ 中的一个元素, 然后取每一个串的前 k 个单词; 如果连接后的串不足 k 个单词, 那么整个串就置于结果集中。因而, 若 $First_2(u)$ 是集合 $\{ab, cd, d, dd, \epsilon\}$, 且 $First_2(v)$ 是集合 $\{cc, d, \epsilon\}$, 则 $First_2(uv)$ 的构造方法是将 $First_2(u)$ 中的 5 个串逐一与 $First_2(v)$ 中的 3 个串连接, 在产生的 15 个串

$abcc\ abd\ ab,\ cdcc\ cdd\ cd,\ dcc\ dd\ d,\ ddcc\ ddd\ dd,\ cc\ d\ \epsilon$

中取每个串的前两个字符, 删除重复元素后即得集合 $\{ab, cd, dc, dd, d, cc, \epsilon\}$ 。

规则 F.2 表明, 一个非终结符的 $First_k$ 是左部为该非终结符的所有产生式的右部的 $First_k$ 求并集。规则 F.3 表明, 一个单词自身的 $First_k$ 是一个由该单词单独组成的集合。推而广之, 一个由 k 或少于 k 个单词组成的串的 $First_k$ 是一个由该串本身组成的单元集。规则 F.4 形式上允许了在 $First_k$ 中出现串长小于 k 的串。

例如, 考虑一个简单的文法 G_{20} :

$A \rightarrow Ba \quad B \rightarrow b \quad B \rightarrow c$

根据规则 F.1, 串 Ba 的 $First_1$ 等于 $First_1(First_1(B)First_1(a))$ 。根据规则 F.2, $First_1(B)$ 是 $First_1(b)$ 和 $First_1(c)$ 的并集; 根据规则 F.3, 它们分别是 $\{b\}$ 和 $\{c\}$ 。因而, $First_1(B)$ 是集合 $\{b, c\}$, 从而 $First_1(Ba)$ 是 $First_1(\{ba, ca\}) = \{b, c\}$ 。

表 4-2 中的规则 F.5 构造了一个非终结符 A 的 $Follow_k(A)$ 。该规则意味着若要构造 $Follow_k(A)$, 应搜索文法中那些 A 出现在其右部的所有产生式, 将 A 右边的所有串的 $First_k$ 添加到 $Follow$ 集中, 包括 $Follow_k(B)$, 其中 B 是产生式左边的非终结符。

像 $First$ 和 $Follow$ 这种递归定义的集合可能导致存在多个解的逻辑问题, 我们通过规定

First 和 *Follow* 是最小解的集合来避免这些问题。

据定义, 目标符号 *S* 的 *Follow* 集总包含一个特殊符号 \perp , 该符号表示一个串的结束 (亦即 PDA 的输入结束); 这意味着串结束符总能跟随整个生成的串, 这是一个显而易见、却容易被忽视的事实。对一个编译程序而言, 串结束符表示一个源文件的结束; 在操作系统中, 命令行解释程序的语法分析程序可能将输入行的结束符作为一个串结束符。串结束符 \perp 的长度被假定为与 k 个单词的长度相同, 因而可完全填满它的 $Follow_k$ 串。由于语言中的每一个串都隐式地跟着一个 \perp , 并且 $Follow_k$ 集是递归定义的, 以便在有需要时返回目标符号寻找串结束符, 所以可推知 $Follow_k$ 集仅由长度为 k 个符号的串组成。*Follow* 集不会是空集, 也不会包含空串。

作为 *Follow* 的一个例子, 考虑一个简单的文法 G_{21} :

$$\begin{array}{ll} S \rightarrow Bx & A \rightarrow aA \\ B \rightarrow yAzA & A \rightarrow b \end{array}$$

为计算 *A* 的 $Follow_1$, 我们找出非终结符 *A* 出现在产生式右部的所有产生式, 共有 3 个。其中一个后面跟随着 z , 故 z 属于 $Follow_1$ 集; 另外两个位于各自产生式的最右端, 因而分别关注各自的左部 (即 *A* 和 *B*) 的 $Follow_1$ 集, 并且将它们放入 $Follow_1(A)$ 集。易见 *B* 的 $Follow_1$ 集是 $\{x\}$, 因为 *B* 仅出现在一条产生式的右部, 并且它后面跟着一个 x ; 因而, x 也属于 *A* 的 $Follow_1$ 集。根据 *A* 出现在最右端的另一产生式可发现 *A* 也是其左部, 但右递归非终结符对 *Follow* 集不起作用, 故可忽略之。因而本例中 $Follow_1(A)$ 集是集合 $\{x, z\}$ 。如果文法中还有产生式

$$B \rightarrow AA$$

则根据规则 F.2, $Follow_1(A)$ 将包含 $First_1(A)$, 在本例中即集合 $\{a, b\}$ 。

再回到文法 G_{19} , 该文法定义了若干 a 后面接着数目相同的 b 的所有串; 让我们构造它的 $First_1$ 和 $Follow_1$ 集:

$$\begin{array}{ll} S \rightarrow aSb & First_1(aSb) = First_1(First_1(a)First_1(Sb)) = \{a\} \\ S \rightarrow \epsilon & First_1(\epsilon) = \{\epsilon\} \end{array}$$

$First_1(Sb)$ 递归地调用我们正在求的 $First_1(aSb)$, 但不管该集合最终计算结果是什么, 它将连接到一个 a 的右边, 且连接后的串组成的集合的 $First_1$ 只有单个 a (无视该递归集)。一般而言, 如果 x 是一个单词, 则 $First_1(xy)$ 是 $\{x\}$, 而不用管 y 是什么。

该文法的惟一非终结符是 *S*, $Follow_1(S)$ 包含两个元素 $\{\perp, b\}$ 。由于 *S* 是目标符号, 集合中包含了串结束符; 而集合中的单词 b 则源自针对第一条产生式应用规则 F.5:

$$First_1(First_1(b)Follow_1(S))$$

与构造 $First_1(aSb)$ 类似, 由于 $First_1(b)$ 集合中不含比 $k = 1$ 更短的串, 故没必要计算余下的表达式, 因为它对集合的计算不会起作用。

4.3.2 选择集

对文法中的每一产生式 $A \rightarrow w$, 我们构造选择集

$$Select_k(A \rightarrow w) = First_k(First_k(w)Follow_k(A))$$

文法中一个非终结符 *A* 是 $LL(k)$ 的, 如果不存在两个 *A* 的产生式 (即 *A* 作为左部的产生式) 的选择集包含任何相同的元素。一个文法是 $LL(k)$ 的, 如果该文法中的每一非终结符均为 $LL(k)$ 的。

任何 LL(*k*)文法都是 LL(*k* + 1)的，显而易见：如果 *k* 个向前看符号就足以确定任何产生式的选择，那么多关注一个符号也不会降低确定性；然而反之不然。

所有 LL(0)文法均产生有穷的语言。考虑到在一个 LL(0)文法中不存在任何有意义的选择，这一结论是显然的。产生不同串的两条产生式之间的选择不得不取决于输入串中的向前看符号（查看将生成哪一个串）。一个语言能成为无穷的，其惟一方式是使用递归；如果同一非终结符的产生式中不存在其他（非递归的）选项，则递归过程无法终止。因而，一个 LL(0)文法必定产生恰好一个有穷长度的串，亦或不产生任何串。

在文法 *G*₁₉ 中，*S* 有两条产生式，它们的选择集计算如下：

$$S \rightarrow a S b$$

$$Select_1(S \rightarrow a S b) = First_1(First_1(a S b) Follow_1(S))$$
$$= First_1(\{a\} \{\perp, b\})$$
$$= First_1(\{a \perp, a b\})$$
$$= \{a\}$$

$$S \rightarrow \epsilon$$

$$Select_1(S \rightarrow \epsilon) = First_1(First_1(\epsilon) Follow_1(S))$$
$$= First_1(\{\epsilon\} \{\perp, b\})$$
$$= \{\perp, b\}$$

由于 *S* 的两个选择集不存在任何共同元素，因而 *S* 是 LL(1)的，从而文法 *G*₁₉ 是 LL(1)的。

作为另一个例子，考虑第 2 章中简单表达式的上下文无关文法 *G*₂。为简单起见，我们添加一个新的非终结符 *G* 作为目标符号，同时添加一个新产生式显式地描述串的结尾。表 4-3 中构造了该文法的 *First*₁、*Follow*₁ 和 *Select*₁ 集。

表 4-3 文法 *G*₂ 的 *First*、*Follow* 和选择集

	<i>First</i> ₁	<i>Follow</i> ₁	<i>Select</i> ₁
D.0. $G \rightarrow E \perp$	$\{n, (\}$		$\{n, (\}$
D.1. $E \rightarrow E + T$	$\{n, (\}$	$\{\perp, +,)\}$	$\{n, (\}$
D.2. $E \rightarrow T$	$\{n, (\}$		$\{n, (\}$
D.3. $T \rightarrow T * F$	$\{n, (\}$	$\{\perp, +, *,)\}$	$\{n, (\}$
D.4. $T \rightarrow F$	$\{n, (\}$		$\{n, (\}$
D.5. $F \rightarrow (E)$	$\{(\}$	$\{\perp, +, *,)\}$	$\{(\}$
D.6. $F \rightarrow n$	$\{n\}$		$\{n\}$

通常 *First* 集采用自底向上的方式会更容易构造，即从右部的首个符号为终结符的那些产生式入手；在本例中即 *F* 的两条产生式。这样更容易为包含这些非终结符引用的其他产生式计算 *First* 集。

计算产生式 D.4 的 *First* 只需简单地将非终结符 *F* 的所有 *First* 集求并集，但 D.3 却引发了另一问题：只有先求出 *First*(*T*)才可以求 *First*(*T* * *F*)，而 *First*(*T*)却要靠 *First*(*T* * *F*)求出！通过直观分析即可快速解决这一问题。非终结符 *T* 在左边递归，亦即只要我们选择了产生式 D.3，在生成的部分串中 *T* 就会位于左边，尽管每次都会在右边产生 “**F*”；可跳出递归的惟一方式是选择产生式 D.4，形成一串由星号分隔的 *F*。具体地说，由于第一个符号是 *F*，递归的 *T* 不会为 *First*(*T* * *F*)集贡献任何元素。同理，产生式 D.1 也不会给 *First*(*E* + *T*)集贡献任何元素，该集合的元素都源自产生式 D.2。

Follow 集采用自顶向下的方式更容易构造，并且从目标符号 *G* 开始构造。在本例中，串终

终结符作为一个终结符显式地写在文法中，因而不必为 $Follow(G)$ 专门写下什么。但由于该产生式没有其他选择，并且它也不会产生空串，因而不需要有 $Follow$ 集。

通过浏览文法中有 E 出现的产生式右部，很容易找出 E 的 $Follow$ 集：该文法有 3 个这样的 E ，每个后面均跟着一个终结符。这样就简单了， $Follow(E)$ 正好就是这 3 个终结符组成的集合。

非终结符 T 也在产生式的右部出现了 3 次，但其中仅有一次是在后面跟着一个终结符。终结符 “*” 必定在 $Follow(T)$ 中，然而另外两次出现又带来什么结果呢？这两次出现都是在非终结符 T 之后没有任何符号，但 $LL(k)$ 条件要求的不是产生 k 个符号，而只是向前看 k 个符号，不管这些符号是如何产生的。这意味着 T 的 $Follow$ 集必须包含那些可跟在产生式右部（ T 是最右边的符号）之后的所有东西，这反映在规则 R.5 中递归地引用了左部非终结符的 $Follow$ 集；在本例中，这两种情况的左部非终结符都是 E 。因而， $Follow(T)$ 包含了 $Follow(E)$ 中的所有元素，再加上之前提到的 “*”。

类似地， F 的 $Follow$ 集包含 $Follow(T)$ 中的所有元素；除此之外无其他元素，因为 F 仅出现在 T 的产生式的结尾。

到了这一步，就很容易从 $First$ 和 $Follow$ 集构造出选择集。由于所有 $First$ 集均不包含长度短于 k 的串，故无须考虑 $Follow$ 集，选择集恰好就是 $First$ 集。但倒霉的是， E 的两条产生式的选择集包含相同的元素，因而非终结符 E 不是 $LL(1)$ 的；同理，非终结符 T 也不是 $LL(1)$ 的。从上述任一结论可得，文法 G_2 不是 $LL(1)$ 的。该文法既不是 $LL(2)$ 的，也不是 $LL(3)$ 的。通常一个程序设计语言的文法若不是 $LL(1)$ 的，那么对任意 k 而言，它也不是 $LL(k)$ 的。

4.4 左递归

文法 G_2 的关键问题是 E 和 T 的产生式都是左递归的，即 D.1 中产生式左部的非终结符 E 同时也出现在右部的最左端，D.3 中 T 的情况也类似。分析程序无法通过有穷的向前看符号，了解需要应用几次递归之后才选择一条可令递归终止的产生式，如图 4-3 所示。因而，所有左递归的产生式都不是 $LL(k)$ 的。

考虑以下一个简单的左线性文法：

$A \rightarrow Ax$

$A \rightarrow y$

该文法产生的所有串均由 y 开头，后接任意数目的 x 。由于它是左线性的，可写出一个定义了相同语言的正则表达式，根据第 3 章的转换规则 L.2 可得：

yx^*

根据这一正则表达式，可构造一个右线性文法：

$A \rightarrow yB$

$B \rightarrow xB$

$B \rightarrow \epsilon$

该文法不再是左递归的，但产生相同的语言。如果令 x 和 y 表示由终结符和非终结符组成的任何固定

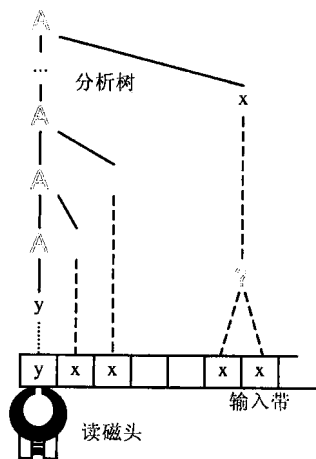


图 4-3 左递归造成的问题。显然，读磁头下的符号 y 必须来自非递归的产生式，跟随其后的每一个 x 必须来自递归产生式的应用；但是从顶部看，在不了解有几个 x 的情况下，无法知道需要应用几次递归产生式

的串，则很容易看出，上述转换规则通过添加一个新的非终结符（如 B 所示），可将任意左递归的上下文无关文法转换为右递归的形式。然而，新的文法是 LL(k)的吗？

暂时假设 x 和 y 分别为终结符，计算其选择集如下：

	$First_1$	$Follow_1$	$Select_1$
$A \rightarrow yB$	$\{y\}$	$\{\perp\}$	$\{y\}$
$B \rightarrow xB$	$\{x\}$	$\{\perp\}$	$\{x\}$
$B \rightarrow \epsilon$	$\{\epsilon\}$		$\{\perp\}$

显然，B 的 Follow 集包含 A 的 Follow 集（据第一条产生式）；如前所述，第二条产生式不会添加任何新元素到该集合中。接着，可根据这两条产生式的 First 集构造出 Select 集。在第三条产生式中，First 集仅含空串，Select 集与 Follow 集相同。

因而可见，在这种简单情况下，我们成功地使得该文法是 LL(1)的。如果令 x 和 y 表示由终结符和非终结符组成的任何固定的串，并将非终结符 A 嵌入到一个更大的文法中，就需要证明 $First_k(x)$ 与 $Follow_k(A)$ 不存在公共元素。常见的程序设计语言通常就是这个样子，这往往是精心设计的结果。

当我们将上述简单的左线性文法转换为右线性时，我们不再将该文法改造为正则的，尽管得到的正则文法仍是一个正确的上下文无关文法形式。究其原因，是因为该文法可能已经不是 LL(1)的：

$$A \rightarrow yB \qquad B \rightarrow xB$$
$$A \rightarrow y \qquad B \rightarrow x$$

显然，A 的两条产生式以同一终结符 y 开头，B 的两条产生式以同一终结符 x 开头。在一个上下文无关文法中允许空产生式的原因之一，是因为在将大多数左递归转换为 LL(k)时这是必不可少的。

禁止出现左递归的非终结符是所有 LL(k)文法的基本要求，如定理 4.4 所示：

定理 4.4 如果 G 是一个 LL(k)文法，则 G 中不存在左递归的非终结符。

本小节开头部分的讨论给出了该定理的非形式化证明，形式化证明留作练习（参见练习 25）。□

4.5 公共左因子

本小节更深入地讨论一种特殊情况：两条产生式的左边有相同的串。考虑以下一个简单的右线性文法：

$$A \rightarrow xy$$
$$A \rightarrow xz$$

由于两条产生式均以单词 x 开头，因而 x 是两条产生式的选择集中的惟一元素；该文法显然不是 LL(1)的。如果我们打算为该语言书写一个正则表达式，一开始可能会写成：

$xy \mid xz$

然而可使用正则表达式的分配律将它转换为：

$x(y \mid z)$

再将它改写为正则文法：

$$A \rightarrow x B$$

$$B \rightarrow y$$

$$B \rightarrow z$$

此时, B 的每一产生式均以不同单词开头, 这些单词定义了 B 的选择集; 新的文法是 $LL(1)$ 的。类似于对递归的处理, 我们会发现, 如果 x 、 y 和 z 表示由终结符和非终结符组成的任意串, 则只要 $First_k(y)$ 与 $First_k(z)$ 不包含公共元素, 文法的 $LL(k)$ 性质就不会丢失。

将一个非 $LL(k)$ 的上下文无关文法转换为一个 $LL(k)$ 文法的两条转换规则可总结如下:

1. 将任意如以下左列所示的左递归形式转换为如以下右列所示的右递归形式, 其中 B 是一个新的非终结符, x 和 y 是由终结符和非终结符组成的任意串。注意, 若 A 有多条非递归的产生式, 新的非终结符 B 必须添加到每一条产生式的结尾; 类似地, 对 A 的每一条左递归的产生式, 必须为 B 创建一条新的右递归产生式取而代之:

$$A \rightarrow A x$$

$$A \rightarrow y B$$

$$A \rightarrow y$$

$$B \rightarrow x B$$

$$B \rightarrow \epsilon$$

2. 将任意如以下左列所示的公共左因子形式转换为如以下右列所示的形式, 其中 B 是一个新的非终结符, x 、 y 和 z 是由终结符和非终结符组成的任意串:

$$A \rightarrow x y$$

$$A \rightarrow x B$$

$$A \rightarrow x z$$

$$B \rightarrow y$$

$$B \rightarrow z$$

回到文法 G_2 , 应用上述规则可将它转换为 $LL(1)$ 的。留意 E 和 T 都是左递归的, 故引入两个新的非终结符 S 和 P , 并为转换得到的文法 G_{27} 计算新的选择集, 如表 4-4 所示。在转换规则中, 表示重复部分的 x 代表了产生式 D.1 中的 “+ T ”; 规则中的终结串 y 即是产生式 D.2 中的 “ T ”。可以相同方式转换 D.3 和 D.4; D.5 和 D.6 则无需转换。

表 4-4 为从文法 G_2 转换得到的 G_{27} 计算 $First$ 、 $Follow$ 和选择集

	$First_1$	$Follow_1$	$Select_1$
$G \rightarrow E \perp$	$\{ n, (\}$		$\{ n, (\}$
$E \rightarrow T S$	$\{ n, (\}$	$\{ \perp,) \}$	$\{ n, (\}$
$S \rightarrow + T S$	$\{ + \}$	$\{ \perp,) \}$	$\{ + \}$
$S \rightarrow \epsilon$	$\{ \epsilon \}$		$\{ \perp,) \}$
$T \rightarrow F P$	$\{ n, (\}$	$\{ \perp, +,) \}$	$\{ n, (\}$
$P \rightarrow * F P$	$\{ * \}$	$\{ \perp, +,) \}$	$\{ * \}$
$P \rightarrow \epsilon$	$\{ \epsilon \}$		$\{ \perp, +,) \}$
$F \rightarrow (E)$	$\{ (\}$	$\{ \perp, +, *,) \}$	$\{ (\}$
$F \rightarrow n$	$\{ n \}$		$\{ n \}$

新文法也可像以前那样直观地阅读, 只要将新的非终结符理解为 “和” 与 “积” 即可: 一个表达式是一个 “项” 后面接着一个 “和”; “和” 这部分要么是加号后面接着一个 “项” 与另一个 “和”, 要么为空。类似地, 一个 “项” 是一个 “因子” 后面接着一个 “积”; 而 “积” 这部分要么是乘号后面接着一个 “因子” 和另一个 “积”, 要么为空。

二义文法引发了另一类问题。一个二义文法不可能是 $LL(k)$ 的, 因为二义产生式的两个不同选择会产生语言中的同一句子。有可能二义性并不是某一语言本身所特有的, 而只是在编写

文法时弄错了；在这种情况下，可从二义串所有可能的分析中剔除多余的分析树，从而解析二义性。

另一方面，如果文法反映的语言二义性类似 Pascal 语言的可选 **else** 子句这种情况，此时无法将文法改写为无二义的，因而也不可能改写为 $LL(k)$ 的，除非重新定义该语言。

4.6 为上下文无关文法扩展正则表达式运算符

将左递归的或含公共左因子的文法改造为 $LL(k)$ 文法时，使用了正则表达式表示其中间形式。在文法中引入正则表达式运算符是完全可行的，实际上也是非常有用的。这相当于为上下文无关文法扩充了正则表达式中用于表示迭代和选择运算的元符号，其优点是以一种清晰的表示法更直观地表达一个由终结符和非终结符组成的序列，而不是使用递归非终结符的冗长产生式。

根据表 3-5 中将正则文法转换为正则表达式的转换规则，并在转换过程中将产生式右部的每一符号看作一个单词，可得到文法 G_2 的扩展文法 G_{2g} ：

$$E \rightarrow T("+" T)^*$$

$$T \rightarrow F("*" F)^*$$

$$F \rightarrow "(" E ")" | "n"$$

从这一文法更容易看出，一个表达式 E 以一个项 T 开头，后面接着任意数目的、由加号连接的另外的项；一个项是一个或多个用乘号连接的因子；一个因子要么是圆括号中的表达式，要么是单词 n 。

注意，从这一文法开始，我们将终结符写在引号中，以免混淆了终结符和元符号。另外还应注意，通常不可能将一个上下文无关文法转换为单个正则表达式；假如能够成功地转换，就足以说明使用上下文无关文法是没有必要的，并且该语言应可描述为一个正则文法。

扩展后的文法更易书写；稍后将看到，它们可自然地转化为以手工方式编写的分析程序代码。这些文法仍能保持 $LL(k)$ 条件吗？回答是肯定的，但选择集的计算会稍微复杂一些。

有一点很重要，即认识到选择集仅与包含选择的产生式有关系；一个不含任何选择的非终结符自然对任何 k 都是 $LL(k)$ 的，无需考虑其选择集。考察文法 G_2 的扩展版本，乍看起来好像不需要任何选择集，但事实并非如此，因为文法中有三个地方必须作出选择。最明显的是有以下选择运算：一个因子要么是一个在括号中的表达式，要么是一个终结符 n ，必须从中作出选择。还有虽不显眼、但也肯定是一个决策点的，是两个迭代运算符之一：在每一轮迭代中，都必须决定是继续循环，还是终止迭代。

在扩展的上下文无关文法中，每一个选择运算和每一个迭代运算均定义了一个决策点。在任一决策点，都须为每一选项构造其选择集。对于选择运算而言，必须为每一选项构造一个选择集；对于迭代运算而言，则必须为迭代体构造一个选择集，再为其 *Follow* 构造另一选择集。在每一决策点，所有选择集必须无公共元素。

令 w 为一个扩展的上下文无关文法的产生式右部片段，且 x 和 y 是由终结符和非终结符组成的任意串，其中可能含有正确嵌套的正则表达式元符号：

$$\begin{aligned} S.1 \quad & \text{如果 } w = x | y & \text{Select}_k(x) = \text{First}_k(\text{First}_k(x) \text{ Follow}_k(w)) \\ & & \text{Select}_k(y) = \text{First}_k(\text{First}_k(y) \text{ Follow}_k(w)) \end{aligned}$$

S.2 如果 $w = x^*$ $Select_k(repeat) = Select_k(x) = First_k(First_k(x) Follow_k(w))$
 $Select_k(quit) = Follow_k(w)$

观察一下 x 和 y 可由什么样的串组成。在选择运算中, 最多只有一个选项的 *First* 集可包含 ϵ , 因为空串允许选择集加入了 *Follow* 集中的元素, 并且如果有多于一个的选项的 *First* 集包含了 ϵ , 则它们的选择集会出现公共元素。类似地, 一个迭代运算的迭代体的 *First* 集不可包含 ϵ , 因为这样会在选择集中加入整个 *Follow* 集, 而这些选择集是不能有公共元素的。

通常很快就可判断一个串是否可空的 (即该串是否可以产生一个空串), 从而更容易发现一个扩展文法不是 LL(1) 的。乘号和问号运算符括住的任何串都是可空的, 带可空选项的任何选择运算也是可空的。两个可空串的连接是可空的; 但如果两个串都不是可空的, 则连接结果也不是可空的。一个单词不是可空的。一个非终结符是可空的, 仅当其产生式 (指扩展文法中的单条产生式) 右部是可空的, 否则该非终结符不是可空的。

回顾文法 G_{28} , 可构造以下选择集, 看看它是否依然是 LL(1) 的:

$E \rightarrow T("+" T)^*$ 决策点: 迭代
 $Select(body) = \{ "+" \}$
 $Select(exit) = Follow(("+" T)^*)$
 $= Follow(E)$
 $= \{ ")", \perp \}$

$T \rightarrow F("*" F)^*$ 决策点: 迭代
 $Select(body) = \{ "*" \}$
 $Select(exit) = Follow(("*" F)^*)$
 $= Follow(T)$
 $= \{ "+" \} \cup \{ ")", \perp \}$

$F \rightarrow "(" E ")" | "n"$ 决策点: 选择
 $Select(left) = \{ "(" \}$
 $Select(right) = \{ "n" \}$

由此可见, 每一决策点的选择集均无公共元素, 该扩展文法确实是 LL(1) 的。注意, 这里构造的一对选择集与为扩展前的文法所构造的相同; 这令人确信一个上下文无关文法在引进正则表达式运算符后, 无论是文法所定义的语言, 还是实现文法的确定 PDA 在识别语言时须援用向前看符号的决策点, 都不会发生本质的改变。

4.7 使用分析程序生成工具

证明一个文法是否 LL(1) 的所有步骤都可机械地执行, 因而也可机械地将一个文法转换为一个确定的 PDA。一个机械化执行该过程的程序称为分析程序生成工具。已有几款分析程序生成工具用于编译程序构造的教学和生产。本书第 10 章介绍了一个特定的分析程序生成工具“TAG 编译程序”的结构。这是一个“编译程序—编译程序”, 即它接受一个扩展文法 (还进一步扩展了本书稍后将介绍的树变换和属性) 作为源语言, 产生一个 Modula-2 或 Pascal 语言源程序实现的分析程序作为输出。借助于属性语义动作, 就有可能用一文法给出一个完整编译程序的规格说明, 然后使用 TAG 编译程序编译该文法。

4.7.1 使用 TAG 编译程序

TAG 编译程序接受的输入文法与我们一直在使用的文法在形式上略有不同。在书写同一非终结符的多条产生式时，我们会在产生式的左部多次重复该非终结符，而由换行符表示一条产生式的结束；而 TAG 编译程序的源语言与许多现代程序设计语言一样，换行符并没有特殊意义，因而必须采用其他方式分隔产生式。单个非终结符的所有产生式必须出现在一起，并且只有其中的第一条产生式指定该非终结符。每一非终结符的最后一条产生式以分号结束。

一个非终结符可以是以字母开头、包含字母和数字的任意标识符，少量保留字除外。除在 **scanner** 部分以正则表达式显式定义的少数终结符之外，其他终结符均用引号括住。类似 Pascal 语言，文法中的注释用花括号括住。文法必须以保留字 **tag** 开头，后面跟着文法的名字和一个冒号；该名字在文法的结束处再次出现，它的前面是保留字 **end**，后接一个英文句号。附录 B 给出了 TAG 编译程序可接受的文法的完整规格说明。

代码清单 4.1 将文法 G_{28} 改写为 TAG 编译程序可接受的形式，使用表示整数的单词 **NUM** 代表 n 。代码清单 4.2 给出了 TAG 编译程序可接受文法的简明文法，它本身就采用了 TAG 编译程序可接受的形式来书写。

代码清单 4.1 文法 G_{28} 的 TAG 编译程序可接受形式

```

tag G2:

scanner

    ignore
        ->      " ";                                { 仅忽略空格 }

    NUM
        ->      ("0" .. "9")+;                       { 定义整数单词 }

parser

    G
        ->      E ".";                                { 用小数点显式地表示串的结尾 }

    E
        ->      T ("+" T)*;                            { 可使用正则表达式的扩展 }

    T
        ->      F P;                                    { 也可使用右递归形式 }

    P
        ->      "*" F P;
        ->      ;                                        { 用空白表示一条空产生式 }

    F
        ->      "(" E ")" | NUM;

end G2.
```

代码清单 4.2 TAG 编译程序可接受文法的文法

tag TagGrammar:	{ 只含语法部分的一个简明定义 }
scanner	{ ==以下为该文法的扫描程序定义部分 }
ignore	{ --保留字, 用于定义被忽略的字符 }
-> " " ""	{ 忽略空格和行结束符 }
-> "{" " " .. "~" " "};	{ 忽略用花括号分隔的注释 }
CHR	{ --定义扫描程序中用到的字符单词 }
-> ('' ' ' " " " ")	{ 行结束符是两个单引号或两个双引号 }
-> "' (" " .. "&" "(" .. "~") "' { 除单引号外的任何其他符号 }	
-> '"' (" " .. "!" "#" .. "~") '"' ; { 除双引号外的任何其他符号 }	
STR	{ --定义分析程序中用到的字符串单词 }
-> "' (" " .. "&" "(" .. "~")+ "' { 不含单引号的任何串 }	
-> '"' (" " .. "!" "#" .. "~")+ '"' ; { 不含双引号的任何串 }	
ID	{ --定义分析程序中用到的标识符单词 }
-> ("a" .. "z" "A" .. "Z") { 以字母开头 }	
("a" .. "z" "A" .. "Z" "0" .. "9")* ; { 后接字母和数字 }	
parser	{ ==以下为该文法的分析程序定义部分 }
TagGrammar	{ 第一个非终结符是目标符号 }
-> "tag" ID ":"	{ 将文法命名为 ID }
("scanner"	{ 可选的扫描程序定义, 由两部分组成: }
("ignore" ("->" scanre)+ ";")? { 忽略的字符, 以及 }	
(ID ("->" scanre)+ ";")+	{ 带名字的单词, 并以分号结束 }
)?	
"parser" (parsrule)+	{ 分析程序的定义接在扫描程序定义之后 }
"end" ID "." ;	{ ID 必须与开头的文法名字匹配 }
scanre	{ 扫描程序的一个正则表达式是: }
-> scanalt (" " scanalt)* ;	{ 用 " " 分隔的选项 }
scanalt	
-> (scanterm)* ;	{ 一个选项是由项组成的序列 }
scanterm	
-> "(" scanre ")"	{ 一个项是由括号括住的正则表达式, }
-> CHR (".." CHR) ? ;	{ 或是一个字符或字符范围 }
parsrule	
-> ID "->" parsre	{ 分析程序的规则是一条产生式, }
("->" parsre)* ";" ;	{ 后接箭头和产生式右部 }
parsre	
-> parsalt (" " parsalt)* ;	{ 产生式的每一右部均为一个正则表达式 }
parsalt	
-> (parstem)* ;	{ 注意: 分析程序的选项可以为空 }

```

parsterm
-> ID                                { 注意：多个 ID 不可直接重复出现， }
-> parsfact ("*" | "+" | "?" | ); { 它们必须出现在 parsfact 的括号中 }

parsfact
-> "(" parsre ")"                    { 要么是一对括号中的正则表达式， }
| STR;                              { 要么是一对引号中的单词串 }

end TagGrammar.

```

4.7.2 使用 YACC

在实际应用中，最流行的分析程序生成工具具有一个略显古怪的名字：Yet Another Compiler Compiler，通常简写为小写的 `yacc`。尽管 YACC 采用简化的自底向上分析算法，而不是本章讨论的有更多限制的自顶向下算法，但是因其应用广泛而值得在此一提。

LL(1)文法是 YACC 所接受的 LR 文法类集的真子集，因而根据本章规则构造的任一正确的 LL(1)文法均可由 YACC 接受；YACC 还可接受某些带左递归和公共左因子的文法，而 LL 算法却不能接受这类文法。如果 YACC 不能接受某一文法，它提供的拒绝理由从自顶向下的角度看可能是毫无意义的，但可以肯定的是该文法不是 LL(1)的。YACC 无法处理本书后续两章所介绍的用于约束检查和代码生成的上下文无关文法扩展。为解决这一类问题，YACC 强制要求编译程序设计人员编写 C 语言的源代码片段。然而在目前的学习阶段，这还不成问题。

从语法上看，YACC 文法的形式非常类似本书使用的 TAG 表示法。扫描程序的定义是分开定义的（使用程序 LEX），带名字的扫描程序单词在 YACC 中采用指示符 `%token` 显式地声明。YACC 使用单个冒号“:”分隔产生式规则的左部（非终结符的名字）和右部，而不是采用右箭头“→”。多条规则合并成单个右部，其间用选择运算符“|”分隔（TAG 编译程序也可接受的形式）；也可以让产生式左部的非终结符名字重复出现多次。终结符用单引号括住，非终结符的引用是简单的标识符，这一点与 TAG 编译程序亦相同。每一产生式也同样以分号表示结束。

此处的目的并不是提供一份 YACC 的教程，任何支持 YACC 的系统都提供了丰富的文档帮助用户学习如何使用它；这里的目的是向读者揭示一些细微的差别，令本章的内容更加容易移植。

4.8 递归下降分析程序

虽然本书的重点是使用合适的工具根据一个上下文无关文法机械地构造分析程序，但 LL(1)文法在以手工编程方式实现一个编译程序时还有一个突出的优点，就是将一个扩展的 LL(1)上下文无关文法转换为诸如 Modula-2 这类传统程序设计语言的递归代码时存在着——映射。这些转换规则相当简单：

文法中的每一非终结符在程序中恰好表示为一个无参数的过程，习惯上采用与非终结符本身相同的标识符命名这一过程。主程序调用目标符号对应的过程。主程序还将初始化扫描程序，并将向前看符号读入到一个全局变量，习惯上该全局变量命名为 `NextToken`。一个非终结符的所有产生式以选择运算合并为扩展文法形式的单条产生式，这一产生式右部形成了实现该非终结符的过程体代码的基础。目前暂时还无需定义过程中的局部变量。

对扩展形式的产生式右部中的每一个元素或结构，恰好对应 Modula-2 语言（或 Pascal 语言，两者的区别微不足道）程序中的一个元素或结构。文法中的每一个非终结符表示为对同名过程的调用；每一终结符表示为对扫描程序的调用。但由于待读入的下一单词已存放在向前看符号中，分析程序必须先将向前看符号与文法中的终结符匹配，然后用一个新的向前看符号取代旧的向前看符号。对任一终结符 x ，其形式为：

```
IF NextToken = x THEN
    GetToken
ELSE
    Error
END
```

对每一个选择运算 $x|y$ ，其中 x 和 y 是任意串，必须测试 x 和 y 的选择集：

```
IF NextToken IN Select(x) THEN
    x
ELSIF NextToken IN Select(y) THEN
    y
ELSE
    Error
END
```

当然，选择集是程序员预先计算好的常量集，不必真的写成上述函数调用形式。

对每一个迭代运算 x^* ，必须计算选择集以证明文法是 LL(1) 的，但只有 x 的选择集会用于程序代码中：

```
WHILE NextToken IN Select(x) DO
    x
END
```

在上述这些结构中， x 均表示串 x 根据相同规则转换成的 Modula-2 语言代码。一个空串将翻译为无任何代码，尽管在这种情况下可能需要检查 *Follow* 集以准确地发现错误。

例如，考虑代码清单 4.1 所示的 LL(1) 文法 G_{28} ，代码清单 4.3 给出了一个实现同一文法的、采用 Modula-2 语言编写的递归下降分析程序（其中未展示扫描程序的过程 *Getoken*）。

在 Modula-2 代码的许多地方，出现了多次测试向前看单词是否为同一个值；这是从文法非常机械地翻译为 Modula-2 代码的结果。通常一个带优化功能的编译程序（或者一位勤奋且细心的程序员）可消除这些冗余的测试。

代码清单 4.3 文法 G_{28} 的递归下降分析程序

```
MODULE G2;
TYPE
    Token = (Plus, Star, Left, Right, NUM, Dot);
VAR
    NextToken: Token;
FROM Somewhere IMPORT
    Getoken, InitScanner, Error;
```

```
PROCEDURE E;
    PROCEDURE F;
    BEGIN
```

(* 嵌套了 F、P 和 T 以避免向前引用 *)

```

IF NextToken IN Token{Left} THEN                                (* 第一个选项 *)
    IF NextToken = Left THEN
        Getoken                                                  (* 读入单词 "(" *)
    ELSE
        Error
    END;
    E;                                                            (* 递归调用非终结符 E *)
    IF NextToken = Right THEN
        Getoken                                                  (* 读入单词 ")" *)
    ELSE
        Error
    END
ELSIF NextToken IN Token{NUM} THEN                                (* 第二个选项 *)
    IF NextToken = NUM THEN
        Getoken                                                  (* 读入单词 "NUM" *)
    ELSE
        Error
    END
ELSE                                                                (* 非上述两种情况 *)
    Error
END
END F;

PROCEDURE P;
BEGIN
    IF NextToken IN Token{Star} THEN                                (* 第一个选项 *)
        IF NextToken = Star THEN
            Getoken
        ELSE
            Error
        END;
        F;
        P
    ELSIF NextToken IN Token{Plus, Right, Dot} THEN                (* P 的 Follow1 集 *)
                                                                    (* 第二个选项为空产生式 *)
    ELSE
        Error
    END
END P;

PROCEDURE T;
BEGIN
    F;
    P
END T;
BEGIN (* E *)
    T;
    WHILE NextToken IN Token{Plus} DO
        IF NextToken = Plus THEN
            Getoken
        ELSE
            Error
        END;
    END;
    T

```

```

END (* WHILE *)
END E;

BEGIN (* G2 *)
  InitScanner;
  Getoken;
  E;
  IF NextToken = Dot THEN
    (* Accept *)
  ELSE
    Error
  END
END G2.

```

4.9 递归下降分析程序作为下推自动机

本章一开始就形式化地介绍了下推自动机，然后展示了如何从一个 LL(1)的上下文无关文法构造一个递归下降分析程序，接下来将看看这样的程序是否正确地构成一个 PDA。

回顾一下，一个 PDA 是一个七元组，由输入字母表 Σ 、状态集 Q 、变迁集 Δ 、栈字母集 H 、初始栈符号 h_0 、起始状态 q_0 以及一个（可能为空的）终结状态集 F 组成。在分析程序中，输入字母表显然就是扫描程序返回的单词集合。

执行一个 Modula-2 程序的计算机可理解为具有有穷数目的状态，即源程序中语句的行数，更准确地说是对应的机器指令的位置。在任一时刻，仅有一条原子语句是活跃的，程序以一种明确定义的方式从一条语句前进到下一语句。从语句前进到下一语句对应于从状态到状态的变迁。

所有支持递归的程序设计语言必须将返回地址以某种栈的形式保存在内存中，这正是我们所说的 PDA 的栈。因而，栈字母表是状态编号的子集，表示过程调用语句的位置。大多数变迁规则保持栈的内容不变；尽管从形式上要求弹出栈顶符号，但这类规则立即将该符号压回栈中。过程调用语句是例外，它们还压入各自的语句编号以及每一过程的 **END** 语句。**END** 语句不是按源代码中的次序前进到下一语句，而是返回跟随在过程调用（从栈中弹出其地址）后的语句。

初始栈符号是操作系统的命令行解释程序，或者是一个可终止程序在执行完毕后通常返回到的任何程序。该 PDA 在栈为空时停机，起始状态是主程序的 **BEGIN** 语句。

至此，通过展示在一个递归下降分析程序中如何建模 PDA 的每一个形式化组成部分，我们确信采用手工方式正确编码的分析程序就是一个下推自动机。在不损失方便性的前提下，我们再次维护了形式化的正确性。

小结

本章的重点是根据一个上下文无关文法机械地构造一个分析程序的技术和工具。本章首先研究了下推自动机与上下文无关语言之间的关系。据定义，上下文无关语言中的串由相应的上下文无关文法推导出来。本章说明了对任一上下文无关文法，均存在一个不确定的下推自动机接受该上下文无关文法定义的语言。

本章介绍了 LL(k)文法，这是上下文无关文法的一个受限的子集。对任一 LL(k)文法，均存在一个确定的下推自动机，通过构造最左推导且在每一步推导中向前看超过 k 个输入符号，可接受该文法所定义语言中的任意串。也存在不满足 LL(k)性质的文法，其中的许多文法可通过消除左递归和公共左因子转换为 LL(1)。

本章还给出了从 LL(1)文法构造一个分析程序的实用步骤,既可借助于分析程序生成工具自动地完成,也可不加思索地用特定程序设计语言的结构替换文法中的成份,很容易以手工方式编写一个递归下降分析程序。

缩略词

- LL(k) 从左到右 (Left-to-right) 扫描输入串、分析时采用最左 (Leftmost) 规范推导,向前看不超过 k 个输入符号。
- NDPDA NonDeterministic Push-Down Automaton, 不确定的下推自动机。
- PDA Push-Down Automaton, 下推自动机。
- TAG Transformational Attribute Grammar, 变换属性文法, 本书特有的“编译程序—编译程序”的输入语言。

关键术语

ambiguous (二义的) 指无法确定地分析一个文法;亦即在推导过程中的一步或多步,有多于一条的规则可以应用到推导过程。

configuration (格局) 用 (q, w, c) 表示 PDA 的状态为 q , 待输入的未读进符号串为 w , 以及在 PDA 的栈中有一个特定的串 c 。

compiler compiler (编译程序—编译程序) 带有附加语义描述的分析程序生成工具。

decision point (决策点) 扩展的上下文无关文法中,一条重写规则右部的选择运算符或迭代运算符。

extended grammar (扩展文法) 书写文法时,在一条或多条产生式中使用了正则表达式的运算符。

left-factors (左因子) 指两条或多条产生式右部的左端具有相同的串,且产生式左部为同一非终结符,参阅 4.5 节。例如:

$$A \rightarrow 0 B 1 1$$

$$A \rightarrow 0 1$$

left-recursive (左递归) 即一条形如 $A \rightarrow Ax$ 的产生式,参阅 4.4 节。

LL(k) grammar (LL(k)文法) 从该文法可构造一个确定的自顶向下 PDA,该 PDA 只需在输入带上向前看最多 k 个符号。

LL(1) grammar (LL(1)文法) 是 LL(k)文法的最常见形式,也最容易使用程序设计语言手工转换为 PDA。

lookahead (向前看) 即观察输入中的下一个单词,但不读入它。

lookahead set (向前看符号集) 参阅 selection set 条目。

nullable string (可空的串) 是由终结符和非终结符组成的串,作为语言中某一句子的组成部分,可生成一个空串。

parser (分析程序)

- (1) 一个识别器 (语言中所有串的接受器), 输出每一个被接受的输入串的分析或推导结果, 参阅 recognizer 条目。
- (2) 一个语法分析程序。

parser generator (分析程序生成工具) 用于根据一个输入文法 (通常是一个上下文无关文法) 生成语法分析程序的软件工具。

PDA (下推自动机)

deterministic (确定的 PDA) 对任一给定格局, 仅有一种可能的变迁。

nondeterministic (不确定的 PDA) 意味着一个给定的格局可能有多个变迁。

PDA transition (PDA 变迁) 是形如

$$\delta: Q \times (\Sigma \cup \epsilon) \times H \rightarrow Q \times H^*$$

的偏函数, 其中 δ 属于 PDA 变迁规则集 Δ , Q 是状态集, Σ 是语言的字母表, H 是栈字母表。这是一个偏函数, 因为并不是所有状态与字母表符号的组合都定义了一个变迁。

phrase structure (短语结构) 描述单词如何组合在一起, 形成一个语法上正确的程序。

production (产生式) 即重写规则。

recognizer (识别器)

(1) 一个过程, 接受属于某一语言的所有串, 拒绝所有其他的串。

(2) 一个自动机。

recursive-descent (递归下降) 形容一个 LL(1) 文法的分析程序, 其中每一非终结符恰好表示为一个无参数的过程, 文法中的每一终结符则表示为对扫描程序的一次调用。

selection set (选择集) 在 $Select_k(A \rightarrow w)$ 中包含了从非终结符 A 可推导出的所有串的前缀, 这些前缀的长度为 k 。亦称向前看符号集。

semantic analysis (语义分析) 确定一个程序在计算上的含义。

set (集合) 表示对象聚集的一个原始数学概念。

$First_k(w)$ 可从串 w 推导出的、由 k 个或更少单词组成的所有终结字符串组成的集合 (构造该集合的规则可参阅本章 4.3.1 节)。

$Follow_k(N)$ 可跟在非终结符 N 推导出的任意子串之后、由 k 个或更少单词组成的所有终结字符串组成的集合。

$Select_k(A \rightarrow w) = First_k(First_k(w) Follow_k(A))$, 也称为选择集, 包含可从 A 推导出的串的长度为 k 的前缀。

TAG compiler (TAG 编译程序) 是一个“编译程序—编译程序”, 接受一个扩展文法作为源语言, 生成一个分析程序作为输出。

top-down parse (自顶向下分析) 从栈中的目标符号出发, 在每一步的句型中不断重写最左的非终结符, 总是选择可最终导致与输入串匹配的的产生式。

练习

- (a) 说明 P_0 接受串 c , 并拒绝串 $aacb$ 和 abc 。
(b) 说明 P_0' 接受串 $aacbb$ 和 c , 并拒绝串 abc 。
(c) 说明 P_0'' 接受串 $aacbb$ 和 c , 并拒绝串 abc 。
- 将以下空栈停机的 PDA 转换为等价的终态停机 PDA。

$$P = (\{a, b\}, \{q\}, \Delta, \{S, A, B, a, b\}, S, q, \{\})$$

其中:

$$\begin{aligned} \Delta = \{ & \delta(q, \epsilon, S) = (q, aA), \\ & \delta(q, \epsilon, S) = (q, bB), \\ & \delta(q, \epsilon, A) = (q, aA), \\ & \delta(q, \epsilon, A) = (q, bB), \\ & \delta(q, \epsilon, B) = (q, b), \\ & \delta(q, a, a) = (q, \epsilon), \\ & \delta(q, b, b) = (q, \epsilon) \} \end{aligned}$$

- 将以下终态停机的 PDA 转换为等价的空栈停机 PDA。

$$P = (\{a, b, c\}, \{p, q, r\}, \Delta, \{S, A, B, a, b, c, h\}, h, p, \{r\})$$

其中:

$$\Delta = \{ \delta(q, \epsilon, S) = (q, aAc),$$

$$\delta(q, \varepsilon, S) = (q, bB),$$

$$\delta(q, \varepsilon, A) = (q, aA),$$

$$\delta(q, \varepsilon, A) = (q, bB),$$

$$\delta(q, \varepsilon, B) = (q, b),$$

$$\delta(q, a, a) = (q, \varepsilon),$$

$$\delta(q, b, b) = (q, \varepsilon),$$

$$\delta(q, c, c) = (q, \varepsilon),$$

$$\delta(p, \varepsilon, h) = (q, Sh),$$

$$\delta(q, \varepsilon, h) = (r, h) \}$$

4. 将以下上下文无关文法分别转换为一个不确定的 PDA:

$$(a) S \rightarrow 01A0$$

$$(b) S \rightarrow AacB$$

$$S \rightarrow 0B1$$

$$S \rightarrow ab$$

$$A \rightarrow 01A$$

$$A \rightarrow Cba$$

$$A \rightarrow 01$$

$$B \rightarrow bc$$

$$B \rightarrow 0B1$$

$$C \rightarrow cAbB$$

$$B \rightarrow 0S1$$

$$C \rightarrow c$$

5. 给出以下 PDA 在识别下列串时的每一步骤:

$$P = (\{0, 1\}, \{S, A, B, C\}, \Delta, \{S, A, B, C, 0, 1\}, h, S, \{C\})$$

其中:

$$\Delta = \{ \delta(S, 0, h) = (A, h),$$

$$\delta(S, 1, h) = (B, h),$$

$$\delta(A, 0, h) = (S, h),$$

$$\delta(A, 1, h) = (C, h),$$

$$\delta(B, 0, h) = (C, h),$$

$$\delta(B, 1, h) = (S, h) \}$$

$$(a) 0011001$$

$$(b) 1100110$$

6. 给出以下 NDPDA 在识别下列串时的每一步骤:

$$P = (\{a, b, c\}, \{q\}, \Delta, \{S, A, B, C, a, b, c\}, S, q, \{\})$$

其中:

$$\Delta = \{ \delta(q, \varepsilon, S) = (q, aAbc),$$

$$\delta(q, \varepsilon, S) = (q, SbB),$$

$$\delta(q, \varepsilon, A) = (q, aAb),$$

$$\delta(q, \varepsilon, A) = (q, Babb),$$

$$\delta(q, \varepsilon, B) = (q, BbC),$$

$$\delta(q, \varepsilon, B) = (q, bC),$$

$$\delta(q, \varepsilon, C) = (q, c),$$

$$\delta(q, \varepsilon, C) = (q, \varepsilon),$$

$$\delta(q, a, a) = (q, \varepsilon),$$

$$\delta(q, b, b) = (q, \varepsilon),$$

$$\delta(q, c, c) = (q, \varepsilon) \}$$

$$(a) aabcbabbcb$$

$$(b) abcbcbabbcb$$

7. 为以下文法构造 $First_1$ 、 $Follow_1$ 和 $Select_1$ 集, 并指出该文法是否 LL(1) 的。

(a) $S \rightarrow 0AS$	(b) $A \rightarrow aB$	(c) $S \rightarrow +B$	$G \rightarrow dG$
$S \rightarrow 10$	$A \rightarrow bC$	$S \rightarrow -B$	$G \rightarrow eC$
$A \rightarrow 1$	$B \rightarrow b$	$S \rightarrow dA$	$G \rightarrow \varepsilon$
$A \rightarrow 0SA$	$B \rightarrow aA$	$B \rightarrow dA$	$C \rightarrow +H$
	$B \rightarrow bD$	$A \rightarrow dA$	$C \rightarrow -H$
	$C \rightarrow aD$	$A \rightarrow .F$	$C \rightarrow dD$
	$C \rightarrow bA$	$A \rightarrow eC$	$H \rightarrow dD$
	$C \rightarrow a$	$A \rightarrow \varepsilon$	$D \rightarrow dE$
	$D \rightarrow aC$	$F \rightarrow dG$	$D \rightarrow \varepsilon$
	$D \rightarrow bB$		$E \rightarrow \varepsilon$

8. 为以下文法构造 $First_2$ 、 $Follow_2$ 和 $Select_2$ 集, 并指出该文法是否 LL(2) 的。

(a) $A \rightarrow aB$	(b) $A \rightarrow aAa$
$A \rightarrow bC$	$A \rightarrow bAb$
$A \rightarrow b$	$A \rightarrow aa$
$B \rightarrow aD$	$A \rightarrow bb$
$C \rightarrow aB$	$A \rightarrow a$
$C \rightarrow bC$	$A \rightarrow b$
$C \rightarrow b$	
$D \rightarrow aE$	
$D \rightarrow a$	
$E \rightarrow aB$	
$E \rightarrow bC$	
$E \rightarrow b$	

9. 为以下文法构造 $First_3$ 、 $Follow_3$ 和 $Select_3$ 集, 并指出该文法是否 LL(3) 的。

(a) $S \rightarrow aSc$	(b) $P \rightarrow P"&"P$
$S \rightarrow aAb$	$P \rightarrow P"V" P$
$S \rightarrow cB$	$P \rightarrow P">" P$
$A \rightarrow abA$	$P \rightarrow P"=" P$
$A \rightarrow a$	$P \rightarrow P"-" P$
$B \rightarrow bB$	$P \rightarrow "P"$
$B \rightarrow bcB$	$P \rightarrow "Q"$
	$P \rightarrow "R"$

10. 为以下文法构造合适的 $First$ 、 $Follow$ 和 $Select$ 集, 从而确定需要几个向前看符号 (亦即该文法是否 LL(1)、LL(2) 等)。

(a) $A \rightarrow \varepsilon$	(b) $A \rightarrow 01B$
$A \rightarrow 10B$	$A \rightarrow 0C$
$B \rightarrow A11$	$B \rightarrow 0C10$
$B \rightarrow 0$	$C \rightarrow 10D$
	$D \rightarrow 01$
	$D \rightarrow 0$

11. 消除以下文法中的所有左递归和公共左因子。

(a) $S \rightarrow SaA$	(b) $S \rightarrow aA$	(c) $S \rightarrow Aa$	(d) $S \rightarrow Ab$
-------------------------	------------------------	------------------------	------------------------

$S \rightarrow b B$	$S \rightarrow b B$	$S \rightarrow b$	$S \rightarrow B a$
$A \rightarrow a B$	$A \rightarrow b A$	$A \rightarrow S B$	$A \rightarrow a A$
$A \rightarrow c$	$A \rightarrow b B$	$B \rightarrow a b$	$A \rightarrow a$
$B \rightarrow B b$	$B \rightarrow c B$		$B \rightarrow a$
$B \rightarrow d$	$B \rightarrow c$		
(e) $S \rightarrow A 0$	(f) $S \rightarrow 0 A$	(g) $E \rightarrow E + E$	
$S \rightarrow B 1 0$	$S \rightarrow 1 B$	$E \rightarrow E * E$	
$A \rightarrow B 0 B$	$A \rightarrow A B$	$E \rightarrow (E)$	
$A \rightarrow B 1 B$	$A \rightarrow A 1$	$E \rightarrow n$	
$B \rightarrow 0$	$A \rightarrow 0 1$		
$B \rightarrow 1$	$B \rightarrow 0 1$		
	$B \rightarrow 0 0$		

12. 将以下的简单文法转换为一个扩展文法。

$S \rightarrow 0 A$
 $S \rightarrow 1 B$
 $A \rightarrow A 1 0$
 $A \rightarrow 0$
 $B \rightarrow 0 1 S$
 $B \rightarrow 1$

13. 为以下扩展文法构造 $Select_1$ 集, 并指出各文法是否 LL(1) 的。

- (a) $S \rightarrow (a A | b B)^*$
 $A \rightarrow (a a)^* b$
 $B \rightarrow (b a) | a^*$
- (b) $S \rightarrow 0(011)^*1(A10B)$
 $A \rightarrow (01|\epsilon)|(1B)^*$
 $B \rightarrow (01100)^*$
- (c) $R \rightarrow ("+" | "-")? d^+ ("." d^+)? ("E" ("+" | "-")? d d^+)?$

14. 以下文法表示了 Pascal 语言中的二义 if 语句。试写出一个与 Pascal 语言的 if 语句语法相同的无二义上下文无关文法, 即 i 产生相同的语言。注意, 据 Pascal 语言的定义, else 子句总是与最内层的 if 语句相匹配。证明你的文法是 LL(1) 的, 因而也是无二义的; 否则说明为什么它不可能是 LL(1) 的。

$G \rightarrow S.$
 $S \rightarrow i x t S e S$
 $S \rightarrow i x t S$
 $S \rightarrow p$

15. 分别给出以下文法的一个例子:

- (a) 是 LL(2) 的, 但不是 LL(1) 的。
 (b) 是 LL(3) 的, 但不是 LL(2) 的。
 (c) 无左递归, 但对于任意 k 都不是 LL(k) 的。

16. 判断以下文法是否 LL(k) 的; 如果是, 请指出 k 的值。

$A \rightarrow y B | z B | y | z$
 $B \rightarrow x B | y B | y | \epsilon$

17. 完成引理 4.1 的证明。

18. 通过构造方式, 以下推自动机的术语证明定理 4.2。

19. 通过构造方式, 以上下文无关文法的术语证明定理 4.2。

20. 采用练习 19 中的构造方法, 为练习 4 所定义的两个语言的并 $L_1 + L_2$ 编写一个上下文无关文法, 并

给出两个串的推导例子。

21. 采用定理 4.3 的证明所示的构造方法, 为练习 4 所定义的两个语言的积 $L_1 \bullet L_2$ 编写一个上下文无关文法。
22. 证明如果 L 是一个上下文无关语言, 则 L^* 也是一个上下文无关语言。提示: 采用构造方式的证明技术。
23. 给出 L^* 中两个串的推导例子, 要求:
 - (a) 首先使用练习 4 (a) 的上下文无关文法, 写出 L^* 的上下文无关文法。
 - (b) 然后使用在 (a) 部分得到的 L^* 给出两个串的推导例子。
24. 练习 22 的结论是否表明如果 L 是 $LL(1)$ 的, 则 L^* 也是 $LL(1)$ 的? 证明你的判断。
25. (定理 4.4) 证明如果 G 是一个 $LL(k)$ 文法, 则 G 不会有左递归的非终结符。提示: 证明该定理的逆否命题, 即如果 G 有左递归的非终结符, 则 G 对任意 k 都不是 $LL(k)$ 的。
26. 证明定理 4.4 的逆命题不成立。提示: 假设逆命题为真, 再找出一个反例。
27. 证明如果对上下文无关文法 G 的每一非终结符 A , 向前看符号集 $Select_k(A \rightarrow w)$ 是不相交的, 则文法 G 是 $LL(k)$ 的。
28. (二义性规则) 证明如果 G 是一个 $LL(k)$ 文法, 则 G 是无二义的。提示: 采用反证法。
29. 证明二义性规则的逆命题不成立。提示: 练习 26 采用的证明技术也可应用到本练习。

复习小测验

指出下列陈述是否正确。

1. 如果 L 是一个上下文无关语言, 则 L^* 也是一个上下文无关语言。
2. 如果一个上下文无关语言是 $LL(1)$ 的, 则其选择集不必是不相交的。
3. 如果一个 $LL(k)$ 文法的分析程序在当前输入位置的右边向前看 $k-1$ 个输入符号, 则它能够以确定的方式运行。

后面的 3 个问题与以下文法 G 有关:

$$A \rightarrow Ax \mid Ay \mid xB \mid y$$

$$B \rightarrow xyB \mid xxB \mid \epsilon$$

4. 文法 G 是左递归的, 且无公共左前缀。
5. 文法 G 不是左递归的, 但有公共左前缀。
6. 可采用消除公共左因子的方法消除文法 G 中的左递归。
7. 一个下推自动机识别的语言是上下文无关的。
8. 如果 L 是一个上下文无关语言, 则总是可找到一个下推自动机接受 L 。
9. 一个左递归的文法对任意 k 都不是 $LL(k)$ 文法。
10. 以下文法不是 $LL(1)$ 的:

$$A \rightarrow 0A \mid 0B \mid \epsilon$$

$$B \rightarrow 0B \mid 1B \mid 0 \mid 1$$

编译程序实验项目

1. 通过为每一条产生式构造选择集, 证明你在第 2 章编写的上下文无关文法是 $LL(1)$ 的。如果该文法不是 $LL(1)$ 的, 将它转换为 $LL(1)$ 文法。
2. 将你的文法录入到一个计算机文件, 并作为 TAG 编译程序或 YACC 等分析程序生成工具的输入。你可能需要将产生式修改为分析程序生成工具可接受的形式。如果你正确地完成了第 1 步, 文法无需太多的修改即可被分析程序生成工具接受。
3. 编译你的分析程序。

4. 编写几个小程序, 用编译好的分析程序分析它们。说明正确的程序会被你的分析程序接受, 而有语法错误的程序则被拒绝。

进一步阅读

Aho, A.V. & Ullman, J.D. *The Theory of Parsing, Translation, and Compiling: Vol. 1, Parsing*. Englewood Cliffs, NJ: Prentice Hall, 1972.

参阅第 5.1 小节的 LL(k)文法、*First* 和 *Follow* 集。

Cohen, D.I.A. *Introduction to Computer Theory*. NY: Wiley.

参阅第 18 章关于下推自动机与上下文无关文法之间关系的介绍; 特别是参阅第 19 章关于上下文无关语言的介绍。

Dwyer, B. "Improving Gough's LL(1) Lookahead Generator." *ACM SIGPLAN Notices*, Vol.20, No.11 (November 1985), pp.27-29.

采用关系 *begun_by* 的闭包生成每一词汇表符号的 *First* 集, 采用 *ends* 关系的闭包生成每一非终结符的 *Follow* 集。B. Dwyer 在他的算法中 (参阅第 28 页) 采用一个链表表示一条产生式。

Gough, K.J. "A New Method of Generating LL(1) Lookahead Sets." *ACM SIGPLAN Notices*, Vol.20, No.6 (June 1985), pp.16-19.

解释了 B. Dwyer 的方法。K. Gough 所指的向前看符号集即本书所说的选择集。

Graham, S.L., et al. "An Improved Context-Free Recognizer." *ACM Transactions on Programming Languages and Systems*, Vol.2, No.3 (July 1980), pp.415-462.

特别留意第 417~427 页的第 2 小节, 给出了一个 14 行的类 Pascal 代码的新算法, 其中使用了所谓的识别矩阵。

Schreiner, A.T. & Freidman, H.G. *Introduction to Compiler Construction with Unix*. Englewood Cliffs, NJ: Prentice Hall, 1985.

参阅第 3 章关于语言识别与 yacc 的讨论。

Sudkamp, T.A. *Languages and Machines: An Introduction to the Theory of Computer Science*. Reading, MA: Addison-Wesley, 1988.

参阅第 4 章语法分析、第 8 章的下推自动机与上下文无关文法等内容; 特别参阅第 15 章关于 *First* 集、*Follow* 集以及向前看符号集 $LA_k(A)$ 的介绍。

第 5 章 语义分析与属性文法

本章旨在：

- 为上下文无关文法扩展属性和语义规则
- 为 LL(1)递归下降分析程序扩展两种形式的属性
- 介绍约束属性值的断言
- 为演示技术而开发一个属性文法模板和相应的递归下降分析程序
- 使用属性强制规定标识符的预先声明和强类型检查

5.1 简介

给定语法的一个文法规格说明后，在编译程序中正确地分析输入源程序的语法并不困难，但程序设计语言的组成不单只是语法和短语结构。现代程序设计语言要求程序中使用的标识符必须声明，以及对变量名或过程名的使用必须与其声明相一致。正如第 2 章所述，将一个标识符与其声明进行匹配需要一个上下文敏感语言。

与其去招惹“线性有界自动机”这个大麻烦，导致不确定性和不完备的形式化到处泛滥，还不如采用以上下文无关文法和确定的下推自动机为基础的更保守方法，只要能解决最迫切的问题即可。属性文法就此诞生。

5.2 属性文法

在非计算机时代，一个对象或人的属性是描述该对象或人的特性、品质、特征等，例如一个人的幽默感或一个对象的颜色。将分析树的结点视为对象，我们也可位于某一结点的非终结符所指称的句子片段描述某些属性，例如它的数值（假如它是一个表达式）、它所命名的过程或变量在内存中的位置（假如它是一个标识符）、它可见的标识符集合（在诸如 Pascal 或 Modula-2 等支持作用域的语言中）等。一个上下文无关文法定义了串的语法，却没有定义任何属性，而这些属性显然是我们要编译的语言的组成部分。因而，为上下文无关文法扩展属性及其强加给语言必须满足的约束是非常实用的。

一个属性文法是一个三元组 $A = (G, V, F)$ ，它由上下文无关文法 G 、各种不同属性的有穷集 V 以及属性断言（即关于属性的谓词）的有穷集 F 组成。每一属性与文法中的单个非终结符或终结符相关联；每一断言则与单条产生式相关联，因而仅仅引用到与该产生式的左部或右部中的终结符或非终结符相关联的属性。语言 G 中的一个串也在语言 A 中，当且仅当这个串的分析树上终结符和非终结符结点附带的所有属性可令所有的断言成立。约束程序是编译程序的一部分，负责校验所有断言对于当前正被编译的程序是否成立。

例如，考虑一个小型表达式文法：

$$\begin{aligned} E &\rightarrow T + T \mid T \text{ "OR" } T \\ T &\rightarrow \text{num} \mid \text{"TRUE"} \mid \text{"FALSE"} \end{aligned}$$

图 5-1 是该文法应用到串“3 + 4”的结果。每一个项 T 都有一个表示类型的属性 t ，要么是 *int*（如果它是一个整数常量），要么是 *bool*（如果它是一个布尔量）。表达式的非终结符可以

有一个与这两个属性相关的断言：这两个属性必须是相同的。

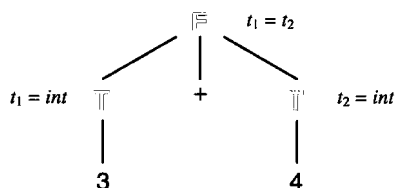


图 5-1 一个简单的带属性的分析树

属性（至少在概念上）是一个静态的值，而不是变量，既可描述终结符，也可描述由属性所属的非终结符所产生的串；但对于语言中不同的串，以及对于这些终结符或非终结符的不同实例，属性的值可能不同。因而属性的值受到引用它的那些断言的约束。断言可能强行指派一个特定的属性值（如上例所示），也可能只是限定属性必须属于某一取值范围。

在理论上讨论一个属性在多个不同数值上的取值可能性或许是有趣的，但在实际上却毫无用处。因而，断言通常分为两类：将一个属性的每个实例约束为单个值的断言称为属性求值函数，没有这么严格的断言则称为谓词。属性求值函数（类似于图 5-1 例子中的断言 $t_1 = int$ ）像赋值语句一样，它们都将属性约束为单个值；而谓词（类似于上述例子中的断言 $t_1 = t_2$ ，假设分析树自底向上阅读）并没有强行指派某一个特定的值，而只是要求两个值必须相等。求值函数约束了一个或多个属性之后，可利用一个谓词给这些属性强加另外的约束；谓词可能约束一个属性必须是某个值（如上例所示），也可能将该属性与一个表达式（表达式中可能涉及其他属性）相关联，例如 $a_1 > a_2 + 3$ 。属性谓词在本质上与谓词演算中的谓词相同，例如谓词 $P(x) = "x \text{ loves Sally}"$ 为在某处定义的某一对象 x 断言：它与一个名为 Sally 的指定对象之间存在着某种关系（恋爱关系）。

在属性文法的定义中通常形式化描述了两类断言；我们认为这种划分是没有必要的，除非是考虑到实现方面的细节。如果上述例子中的分析树是从右到左阅读（而不是自底向上阅读），则断言 $t_1 = t_2$ 是一个求值函数，而断言 $t_1 = int$ 是一个谓词。然而，在一些特定的语境中为了澄清语义问题，我们还是在术语上区分求值函数和谓词。

我们将属性文法写成一个上下文无关文法，再为其中的每个非终结符附加 0 个或多个属性。经典属性文法的表示法采用一种“记录一域”格式，其中对属性的每一个引用均指定了非终结符和属性名，形如“ $N.a$ ”（其中 N 是一个非终结符， a 是一个属性）。留意到属性断言最方便的写法是放在它所作用的产生式的本地位置，并且大多数程序员已熟悉了这种特别的名字与定位表示法的组合表示，因而这样可写出更紧凑、更可读的文法。

属性文法的功能之一是形式化地说明整个文法的上下文信息流，而不是为此建立一个上下文敏感文法。这一点很重要，因为已有高效的算法可根据一个上下文无关文法机械地构造一个下推自动机，但是对于线性有界自动机却缺乏这类工具。不过从理论上可证明，虽然在形式上属性文法并不是一个上下文敏感文法，它却定义了一个上下文敏感语言（尽管可能会有一些限制）。然而属性求值函数和谓词却可直接转换为分析程序自动机的实现，同时不会破坏原有的算法构造过程。

5.2.1 继承属性和综合属性

属性以两种方式出现：继承属性由在右部引用了该属性所附属的非终结符的那些产生式中

的断言定义；综合属性（又称派生属性）则定义在该属性所附属的非终结符的产生式之中，或者就是该属性所附属的终结符本身固有的值。如图 5-2 所示，在分析树中继承属性继承了其父结点的值，在文法中它们用一个向下箭头表示（ $\downarrow attrname$ ）。综合属性可进一步分为两类：一类仅局部地用在定义该属性的产生式中，一类则沿分析树向上传递给父结点；后者在文法中用一个向上箭头表示（ $\uparrow attrname$ ）。在图 5-1 的例子中，如果分析树自底向上求值，则 t_1 和 t_2 均为综合属性。

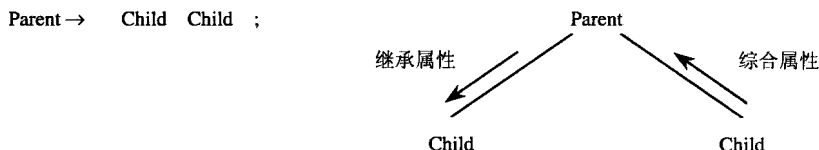


图 5-2 分析树中的继承属性和综合属性，其中展示了属性的信息流

属性文法在 1968 年 Donald Knuth（高德纳）发表的一篇论文中已完全定型，但我们现在才开始在编译程序设计中有效地运用这一技术。尽管如此，高德纳关于属性求值的例子仍是这一概念的最佳教学素材。考虑如下一个定义了定点二进制的上下文无关文法：

$$\begin{aligned} N &\rightarrow S "." S \\ S &\rightarrow S B \\ &\rightarrow B \\ B &\rightarrow "0" \\ &\rightarrow "1" \end{aligned}$$

显然，我们也可写出一个简单的正则表达式定义同样的语言，但此处目的只是展示如何为一个非终结符附加继承属性或综合属性。

目标符号 N 表示整个二进制数，我们为它附加一个综合属性 v ，表示该二进制数的数值：

$$N \uparrow v$$

符号 v 表示非终结符 N 生成的任意二进制数的值。在该语言的任一棵特定的分析树中，从目标符号推导出的终结字符串都将有一个特定的值 v ，根结点 N 就是以这个值作为属性。非终结符 B 表示一个二进制数字，它有它自己的值 v （不要与附加到 N 的同名属性相混淆）。然而，一个数字的值对二进制数 N 整个值的贡献还取决于该数字在二进制数中的位置，该数字的值将根据它与二进制小数点的距离换算为 2 的幂，这个换算系数 f 无法由该数字本身可用的信息综合出来，只能从它的父结点（非终结符 S ）继承下来：

$$B \downarrow f \uparrow v$$

我们可定义一个断言，将值 v 和换算系数 f 相关联。明确地说，当数字为 1 时其值必须等于换算系数；当数字为 0 时换算系数是无所谓的，该数字的值可断言为 0。

非终结符 S 表示一串二进制数字，它也有一个依赖于串中位置的值 v ，是一个关于继承得到的换算系数 f 的函数。但是这一换算系数从何而来？它必须以串长以及相对于二进制小数点的位置为基础。在 N 和 S 的产生式中即可确定位置，然而串长 l 只能在 S 的产生式中逐位构造该串时综合得到。因而，我们不仅为 S 附加一个从根结点（可能是间接地）继承的换算系数，同时也附加了综合属性分别表示串长及其值：

$$S \downarrow f \uparrow v \uparrow l$$

现在可写出每一非终结符的产生式所关联的属性断言。这些断言仅用于构造二进制数的值（亦即将它约束为单个值），因而我们将这些断言称为属性求值函数也是同样正确的：

$$\begin{aligned}
 N \uparrow v &\rightarrow S \downarrow f_1 \uparrow v_1 \uparrow l_1 \text{ " " } S \downarrow f_2 \uparrow v_2 \uparrow l_2 & [v = v_1 + v_2; f_1 = 1; f_2 = 2^{-l_2}] \\
 S \downarrow f \uparrow v \uparrow l &\rightarrow S \downarrow f_1 \uparrow v_1 \uparrow l_1 B \downarrow f_2 \uparrow v_2 & [f_1 = 2f; f_2 = f; v = v_1 + v_2; l = l_1 + 1] \\
 &\rightarrow B \downarrow f \uparrow v & [l = 1] \\
 B \downarrow f \uparrow v &\rightarrow \text{"0"} & [v = 0] \\
 &\rightarrow \text{"1"} & [v = f]
 \end{aligned}$$

正如正则文法中语义动作的写法，断言用方括号括住。为文法产生式右部的非终结符所附加的属性通常采用不同的名字（此处是加上下标），从而在断言中像使用 v 就不会出现引用混乱。

观察文法的第一条产生式，会发现一个二进制数的值 v 是值 v_1 （整数部分）与 v_2 （小数部分）之和，而整数和小数部分通过沿分析树向下传送的换算系数已可正确地换算。二进制数整数部分的二进制数字串继承的换算系数是 1，而小数部分二进制数字串继承的换算系数则取决于其右端与二进制小数点之间的位数，亦即该串的长度。

对于由一个子串后接一个二进制数字组成的任意串 S （第一条产生式），二进制数字 B 的换算系数 f_2 与整个串的换算系数相同，但子串的换算系数则是该值的 2 倍，因为它左移了一个二进制位。整个串的长度比子串长度大 1，整个串的值 v 是子串的值与数字的值之和。当一个二进制串仅由单个数字 B 组成时，换算系数是相同的，并且派生的值也是相同的；在文法中可省略这些断言，只须令这两个非终结符的属性名相同。

最后，如前所述，一个二进制数字的值是该数字乘以它的换算系数；如果该数字为 0，则这个值也为 0。

图 5-3 展示了属性值在该文法的一棵分析树上是如何流动的。注意，尽管图中画作三遍求值，依次在不同时刻计算不同属性的值，但一棵完整的带属性的分析树却是将所有属性附加到单棵树的、属性各自所属的结点上，使得所有的断言可同时成立。该例子的属性求值无法在一遍求值中完成，这可看作是实现阶段的产物。实际上，这一属性文法几乎只是勉强能够求值，求值次序至少需要三次遍历分析树：一遍自底向上求串的长度，一遍自顶向下求换算系数，最后另一遍自底向上求整个值。

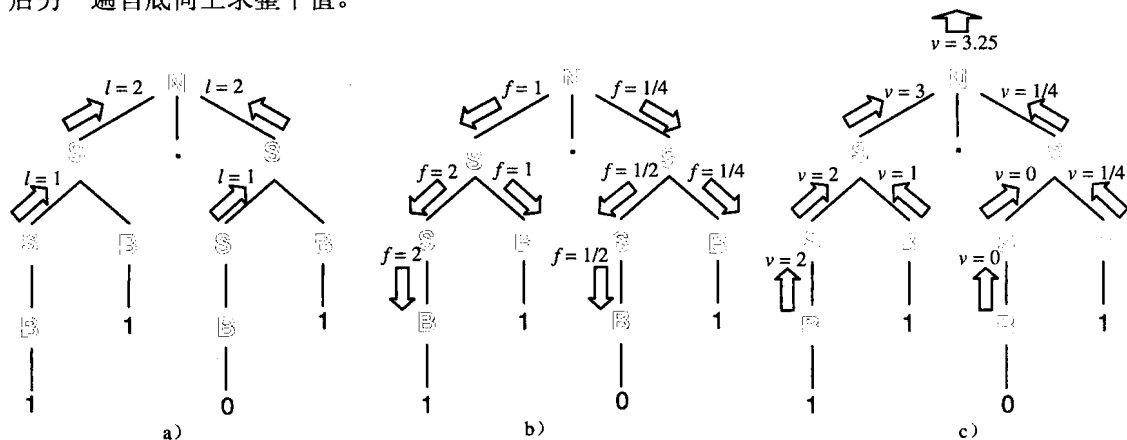


图 5-3 串 11.01 的分析树上的属性流。长度的属性向上流动（图 a）；然后换算系数的属性向下流动（图 b）；最后整个值的属性向上流动（图 c）

可以想像, 属性求值函数也会以一种相互依赖的方式定义。例如, 以下一个小文法中的 u 和 v 、 x 和 y :

$$\begin{array}{ll} A & \rightarrow B \downarrow x \uparrow y \\ B \downarrow u \uparrow v & \rightarrow a \end{array} \quad \begin{array}{l} [x = y] \\ [v = u] \end{array}$$

这种依赖关系称为循环依赖, 因为继承属性 x 依赖于综合属性 y , 而 y 与 v 相同, v 反过来依赖于 u , 而 u 却与 x 相同。已有许多研究探讨了如何避免属性的循环依赖, 以及寻找属性求值次序的高效策略。

在刚开始讨论二进制数求值时我们就强调, 这是一个挖空心思设计的例子, 主要是为了展示综合属性与继承属性的区别。它也提醒了我们属性求值次序这一潜在的问题。在实践中, 我们不会试图编写一个需要如此复杂的属性求值策略的编译程序。例如, 在二进制数的例子中如果选用一个不同的属性集, 一遍自底向上就足以完全求出属性的值。假设从非终结符 S 和 B 中省去属性 f 和 v , 而用单个综合属性 i 取而代之, 表示二进制数字或数字串的整数值 (看作一个整数), 文法即可简化为:

$$\begin{array}{ll} N \uparrow v & \rightarrow S \uparrow i_1 \uparrow l_1 \quad "." \quad S \uparrow i_2 \uparrow l_2 \\ S \uparrow i \uparrow l & \rightarrow S \uparrow i_1 \uparrow l_1 \quad B \uparrow i_2 \\ & \rightarrow B \uparrow i \\ B \uparrow i & \rightarrow "0" \\ & \rightarrow "1" \end{array} \quad \begin{array}{l} [v = i_1 + 2^{-l_2} \cdot i_2] \\ [i = 2 i_1 + i_2; \quad l = l_1 + 1] \\ [l = 1] \\ [i = 0] \\ [i = 1] \end{array}$$

此外, 由于现在仅需一遍自底向上的遍历, 属性求值可与语法分析同时进行。虽然情况并非总能如此美妙, 但我们越是仔细地选用属性求值函数, 就会使得编译程序的工作效率越高。

5.2.2 属性值流

在一个典型的带优化功能的编译程序中, 通常会遇到四类属性求值需求或信息流:

1. 自底向上
2. 自顶向下
3. 从左到右
4. 从右到左

根据正在处理的子树的某一固有性质导出其值的那些属性 (比如二进制数文法中的长度属性 l), 通常采用自底向上的求值次序。而依赖于子树所处的部分上下文或环境的属性, 则通常需要某种形式的自顶向下求值。很少属性会在整个文法中严格地按照自顶向下次序求值, 它们往往受限于编译过程开始之前就已有的编译程序开关或其他全局信息。大多数继承属性依赖于在文法其他地方求值的综合属性, 这一类别通常划分为从左到右信息流和从右到左信息流。从左到右的信息流在分析树中某一特定结点的左子树导出, 再由该结点的右子树继承; 通常属性值由非终结符继承, 在其产生式中 (或在分析树中该产生式的下方) 还可能被修改, 修改后的值导出一个新的属性。

图 5-4 展示了二进制整数的一个二义的上下文无关文法, 并附加了一个从左到右的属性求值流。每一个二进制数字的子串将它自身的值添加到继承属性值的右边, 空子串仅仅传递这个值而不改变它。串 “10011” 的分析树展示了属性值的流动。

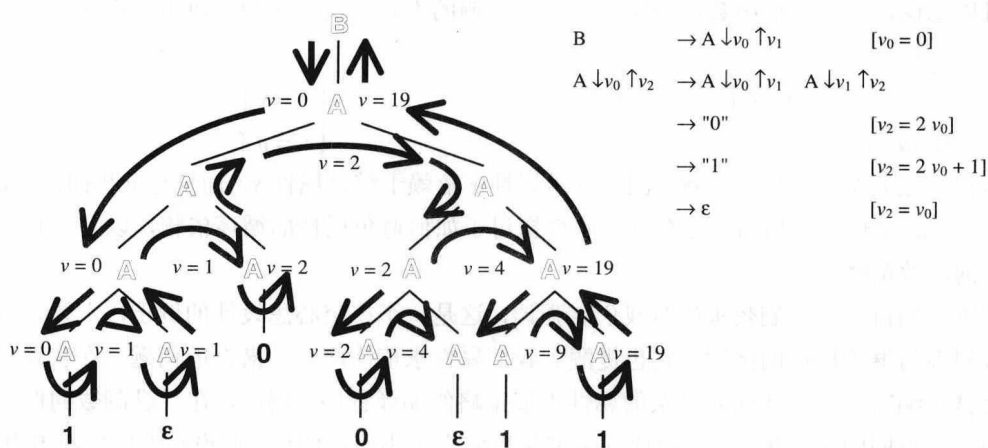


图 5-4 串 10011 从左到右的属性流

本章将深入讨论 Pascal 和 Modula-2 这类语言，这类语言要求标识符在使用前须先声明。标识符的先声明是从左到右的信息流的一个实例，因为变量声明位于变量被引用的左边。第 8 章介绍的优化课题涉及一个变量下次将被如何使用，这显然是一个从右到左的信息流。

自顶向下和自底向上均可在语法分析时对纯综合的（自底向上的）属性进行求值。由于程序是从左到右进行分析的，大多数从左到右的属性求值在自顶向下和自底向上的分析程序中都是可行的，不过在自底向上的分析程序中须仔细处理以保证信息流的一致。一般而言，如果某一属性需要用到在逻辑上位于分析程序向前看符号右边的信息，则该属性将无法在语法分析的同时求值。第 8 章讨论了基于分析程序在内存中所构造的抽象语法树的属性求值，其灵活性远远高于与语法分析同时进行的属性求值，但是也更浪费内存和 CPU 时间。

大多数程序设计语言可沿严格的从左到右、自底向上属性流以一遍方式被编译（虽然生成的代码效率不高）。练习 1(d)和 1(e)演示了属性文法的信息流设计好坏所带来的影响。

5.3 非终结符作为属性求值函数

一旦指派了哪些属性是继承的、哪些属性是综合的，就确立了属性信息流的方向，从而可将属性求值函数收集或分离到文法的空产生式中；这些空产生式的惟一目的就是封装语义。再次考虑以下这个小属性文法：

$$\begin{aligned}
 E &\rightarrow T \uparrow t_1 "+" T \uparrow t_2 & [t_2 = t_1] \\
 T \uparrow t &\rightarrow num & [t = int] \\
 \text{属性谓词 } [t_2 = t_1] &\text{可封装在以一个新非终结符 } X \text{ 开头的另一产生式中：} \\
 E &\rightarrow T \uparrow t_1 "+" T \uparrow t_2 X \downarrow t_1 \downarrow t_2 \\
 X \downarrow t_1 \downarrow t_2 &\rightarrow \epsilon & [t_2 = t_1] \\
 T \uparrow t &\rightarrow num & [t = int]
 \end{aligned}$$

这种写法的效果是显式地指明了它是一个约束两个属性相等的谓词，而不是一个属性求值函数。若要强制规定它是一个属性求值函数，则可直接指派一个不同的信息流：

$$\begin{aligned}
 E &\rightarrow T \uparrow t_1 "+" T \uparrow t_2 X \downarrow t_1 \uparrow t_2 \\
 X \downarrow t_1 \uparrow t_2 &\rightarrow \epsilon & [t_2 = t_1]
 \end{aligned}$$

$$T \uparrow t \quad \rightarrow num \quad [t = int]$$

此时, 第一个非终结符 T 断言 t_1 应为 *int* (通过构造方式), 并且该值向下传递给 X , 而 X 断言 t_2 必须是相同的值。这个值依次沿分析树向上传递给 E , E 则隐式地断言它必须与 t_2 (非终结符 T 的第二个实例已断言它为 *int*) 相同。

尽管这个例子很简单, 但还是容易看出, 对非终结符 X 及其继承属性和综合属性的引用可有效地取代断言 $t_2 = t_1$ 。我们以两种方式利用这种等价性。首先, 它提供了一种简单的方法将语言的约束封装为文法中独立的专有部分, 从而使文法具有更好的可读性。可根据逻辑相关性将属性断言集中在一起并加以命名, 而不是随机地分散在语法规则中。它还有一个副作用, 就是有助于为特定的属性在谓词或属性求值函数两者之间选择实现方式, 因为信息流在文法的表示中是显式表达的。这种等价性的第二个好处更重要, 因为它为任意复杂的属性求值函数和谓词提供了一种一致的表示法。

我们已在属性断言中使用了常见的算术运算符和比较运算符, 这些运算符具有普通的数学含义。然而, 传统的运算符并不能满足编译程序的所有功能需求; 有一类属性断言无法采用普通运算符表达, 它们涉及符号表的概念。

5.4 符号表作为属性

属性文法的一个主要功能是保证一个由语法上正确的串所组成的语言满足某些语义约束。强类型语言的一个约束是任何变量或子表达式的声明类型或导出类型必须与它们的用法保持一致。另一个常见的约束是任何标识符都必须“先声明、后使用”。在第 4 章编译程序实验项目所开发的语法规则中, 无法强制被编译的程序中标识符必须预先声明, 当时直接跳过了这一问题。强类型约束可加入到你的第一个文法中, 方法是在语言的语法中拒绝任何破坏了类型约束的表达式; 其结果是一种太简单的语言, 而不是真正的 *Pascal* 或 *Modula-2* 语言。现在考虑应为文法引入哪些属性, 才能为该语言的一个更大的子集适当地添加上述两个约束。

经典的符号表是一个将标识符集与值集相关联的数据结构。扫描程序已将每一标识符编码为一个惟一的整数, 在约束程序中将这个值看作是终结符 *ID* 的一个综合属性。另一方面, 符号表中与标识符相关联的值通常是复杂的信息记录。一个标识符须关联的数据包括: 它的类别是什么 (例如类型名、变量名、过程、或常量等), 它在目标机器的内存中驻留何处 (对于变量和过程), 它的类型是什么, 等等。符号表中的每一符号必须在某种意义上是惟一的, 因而符号表可看作是从整数到值记录的一个动态函数。

符号表有两种基本访问模式: 一种是添加一个新的标识符及其值, 并校验该标识符在表中未曾出现; 另一种是查找一个特定的标识符, 并取出检索到的值。我们将这两种访问函数定义为形如非终结符的属性求值函数, 其中符号表、标识符、值记录都是属性; 非终结符引用中的继承属性和综合属性也采用相同的表示法, 但括在方括号中以表明这是一个预定义的属性求值函数, 而不是一个非终结符。符号表访问函数写成如下形式:

```
[into ↓oldsymtab ↓ident ↓value ↑newsymtab]
  { 将 ident:value 加到 oldsymtab 并返回含有 ident 的 newsymtab }
[from ↓symtab ↓ident ↑value]
  { 在 symtab 中查找 ident 并返回它的值 }
```

属性求值函数 **into** (通过构造方式) 断言: **newsymtab** 是一个与 **oldsymtab** 相似的符号表, 惟一区别是它还包含了将 **ident** 与 **value** 相关联的入口; 此外, 该求值函数还断言: 在 **oldsymtab** 中不存在与 **ident** 冲突的符号。类似地, 求值函数 **from** 断言: **ident** 已在符号表 **symtab** 中, 且它与值 **value** 相关联。

注意, 我们一直避免将属性看作可随时修改的变量, 而是看作值, 这些值会根据求值函数 (实际上就是断言) 关联到新的值。因而, 尽管我们很容易将符号表看作某种传统程序设计语言中部分填充的记录数组, 并且事实上在编译程序的数据空间中这一数据结构也可能是一种好的实现, 但是如果让这种表示蒙蔽我们的理解, 将令属性文法带来的形式化与概念化优势烟消云散, 我们也退化到像用 C 这类低级程序设计语言处理编译程序。在实践中, 我们不会真的在每次使用 **into** 时都复制整张符号表, 而只是将新符号添加到顶部, 并返回一个指向符号表新顶部的指针; 如果符号表的实现采用一个记录的链表, 这根本不会导致性能低下。

标识符是分析程序所处理语言中的终结符, 扫描程序识别其形式, 并为任一标识符均报告同一个单词 **ID**; 但标识符声明与类型检查却要求识别出一个个不同的标识符。第 3 章展示了如何向扫描程序文法添加语义动作, 以及如何据此将语义动作添加到一个实现扫描程序的 FSA 中。其中的一个语义动作为字符串表中每一个惟一的标识符产生一个惟一的整型标识符编号。扫描程序返回的标识符编号将作为一个综合属性。注意, 由扫描程序负责维护的字符串表与这里的符号表没有关系, 不应将两者混淆。字符串表包含标识符的拼写, 而符号表则关注抽象属性和符号性质。以往的符号表通常也包含了标识符的拼写 (作为符号的另一性质), 但这在现代程序设计语言中是不实用的, 因为标识符长度不再像早期编译程序那样限制在 6 个或 8 个字符。

5.5 Micro-Modula 的属性文法

本小节为第 2 章 2.2.3 小节的文法 G_2 扩展了变量声明和赋值语句, 并添加了一些布尔运算使得类型检查更有意思, 从而得到如代码清单 5.1 所示的 Micro-Modula 文法。

代码清单 5.1 Micro-Modula 语言的语法文法

```

M
->  "MODULE"  ID  "; "
    "VAR"
      V
    "BEGIN"
      B
    "END"  ID  ". "  ;

V                                     { 0 个或多个变量声明 }
->  ID  ":"  T  "; "
    V
->  ;

T                                     { 两个预定义类型 }
->  "INTEGER"
->  "BOOLEAN"  ;

B                                     { 仅在模块体中有赋值语句 }
->  ID  " :="  E  "; "
    B

```

```

-> ;

E                                     { 简单表达式, 或是 ... }
-> S C ;

C                                     { ... 比较相等 }
-> "=" S
-> ;

S                                     { 注意, 该文法是 LL(1) 的 }
-> F P ;

P
-> "*" F P
-> "AND" F P
-> ;

F
-> "(" E ")"
-> ID
-> NUM                               { 整型常量 }
-> "TRUE"                           { 布尔型常量 }
-> "FALSE" ;                       { 布尔型常量 }

```

现为该文法附加类型检查所需的属性。首先, 注意到单词 **ID** 有一个综合属性 **idn**, **idn** 惟一地确定了标识符的拼写。目标符号的产生式中有两个标识符, **Modula-2** 语言要求它们必须是相同的标识符, 即它们具有相同的 **idn**。很容易写出一个属性断言强制规定这一点, 要么在产生式中将它们命名为同一名字(正如代码清单 5.2 中的做法), 要么显式地写出一个等式谓词。在文法的其他地方, 均利用标识符的属性以区别符号表中的标识符。注意在代码清单 5.2 中, 向上箭头“↑”录入为 ASCII 的音调符“^”, 向下箭头“↓”录入为感叹号“!”。

代码清单 5.2 Micro-Modula 语言的属性文法, 支持类型检查

```

M !vacantTbl
-> "MODULE" ID ^idn ";"
    "VAR"
    V !vacantTbl ^tblOut
    "BEGIN"
    B !tblOut
    "END" ID ^idn "." ;

V !tblIn ^tblOut
-> ID ^idn ":" T ^type ";" [into !tblIn !idn !type ^nutbl]
    V !nutbl ^tblOut
-> [tblOut = tblIn] ;

T ^type
-> "INTEGER" [type = 1]
-> "BOOLEAN" [type = 2] ;

B !tblIn
-> ID ^idn "==" E !tblIn ^type ";" [from !tblIn !idn ^type]
    B !tblIn

```

```

->      ;

E !tblIn ^type
->      S !tblIn ^stype C !tblIn !stype ^type ;

C !tblIn !typeIn ^typeOut
->      "=" S !tblIn ^ctype [typeIn = ctype; typeOut = 2]
->      [typeOut = typeIn] ;

S !tblIn ^type
->      F !tblIn ^type P !tblIn !type ;

P !tblIn !type
->      "*" F !tblIn ^type P !tblIn !type [type = 1]
->      "AND" F !tblIn ^type P !tblIn !type [type = 2]
->      ;

F !tblIn ^type
->      "(" E !tblIn ^type ")"
->      ID ^idn [from !tblIn !idn ^type]
->      NUM ^value [type = 1]
->      "TRUE" [type = 2]
->      "FALSE" [type = 2] ;

```

我们还需要一个继承的（从左到右的）符号表属性在整个文法中携带信息。假设目标符号继承了一个空的符号表属性 **vacantTbl**，且每一非终结符继承一个 **tblIn**；变量声明的非终结符也由 **tblIn** 导出一个（通常是修改过的）**tblOut**，并作为其右边的下一非终结符所继承的 **tblIn**。类似地，类型检查还强制规定了表达式内部采用从左到右的属性求值次序。

考虑代码清单 5.2 中的属性文法。表达式 **E** 由简单表达式 **S** 后接比较运算 **C** 组成，**C** 可能是以 “=” 分隔的另一简单表达式，也可能为空。当 **C** 为空时，表达式 **E** 的类型与作为其组成部分的简单表达式 **S** 的类型相同，但这在对 **E** 的产生式只进行一遍分析时是无法得知的；因而将这一类型值向下传递给 **C**，然后在产生式 $C \rightarrow \epsilon$ 原封不动地返回它。另一方面，如果 **C** 是一个比较运算的余下部分，那么两个子表达式的类型必须相同，不是整型就是布尔型（文法中的等式谓词表达了这一点），并且比较运算的结果类型是布尔型。

简单表达式 **S** 的类型与作为其组成部分的因子 **F** 的类型相同，于是综合属性原封不动地向上传递；同时，它还必须与因子右边的乘积组成部分 **P** 是同一类型。在乘积 **P** 的产生式中，如果乘积为空则不必理会其类型；但如果乘积部分有乘法运算，则因子必须推出与乘积部分继承得到的相同类型；递归的乘积部分继承同一类型，且该类型必须是整型（此处编码为数字 1）。如果运算符是 **AND**，则所有涉及的类型必须是布尔型（编码为 2）。

常量因子的类型被综合为一个常量（取值 1 或 2，取决于常量是一个数值还是关键字 **TRUE** 或 **FALSE** 之一）；而变量的类型则必须从符号表中查找。

变量在产生式 **V** 中声明，其中每一个标识符要么与关键字 **INTEGER** 关联（从而推出类型属性为 1），要么与关键字 **BOOLEAN** 关联（从而推出类型属性为 2）。标识符及其类型被登记到继承下来的符号表中，新符号表再递归地传递给该声明右边的任何声明，从而一直向上传递到程序体，在这里被赋值语句及其成分表达式所继承。

处理赋值语句体时，先在符号表中查找赋值运算符左边的标识符，再断言查表返回的类型

必须与右边表达式推出的类型相等。文法通过引用相同的属性名，隐式地表达了这一约束。

5.6 在 TAG 编译程序中使用属性

TAG 编译程序被设计为根据一个属性文法的源文件生成合适的约束检查代码。然而，不同于之前书写的小文法，TAG 编译程序是强类型的。继承属性和综合属性必须在产生式头部用一个类型名声明，非常类似 Modula-2 和 Pascal 语言要求的变量声明。

此时涉及的两个预定义类型是 **int**（整型数字）和 **syntab**（符号表）。内置属性求值函数也必须声明其属性类型；TAG 的 **scanner** 部分使用的内置语义求值函数以相同的方式预声明。与强类型程序设计语言一样，预声明的类型需求使得编译程序能执行合理的一致性检查，从而检测和报告更多常见的编码错误。

代码清单 5.2 中文法的头部信息看起来类似代码清单 5.3。其中有一种特殊的语法，它采用“@”符号在迭代的正则表达式中对属性求值；第 10 章详细解释了这种语法，目前它暂时只用于整型的求值。此外还应注意，文法中的 **VacantTable** 已被一个空表构造算子“<>”取代。

代码清单 5.3 为 TAG 编译程序编写的 Micro-Modula 属性文法的头部

```
tag MicroModula:

predeclared

    into !syntab !int !int ^syntab ;
    from !syntab !int ^int ;
    charval ^int ;                { 扫描程序函数，返回一个字符的序数 }
    initbl ;                      { 扫描程序函数，开始扫描一个标识符 }
    addtbl !int ;                 { 扫描程序函数，将一个字符添加到字符串表 }
    strindex ^int ;              { 扫描程序函数，在表中查找标识符 }

scanner

    ignore
        -> " " | " " ;          { 删除空格和行结束符 }

    ID ^name:int
        -> [initbl]
            ("a" .. "z" | "A" .. "Z") [charval ^this; addtbl !this]
            (("a" .. "z" | "A" .. "Z" | "0" .. "9")
                [charval ^this; addtbl !this])*
            [strindex ^name] ;

    NUM ^value:int
        -> [value = 0]
            (("0" .. "9") [charval ^this; value@ = value * 10 + this - 48]) + ;

parser

    M
        -> "MODULE" ID ^idn " ; "
            "VAR"
                V !<> ^tblOut
```

```

        "BEGIN"
        B !tblOut
        "END" ID ^idn "." ;

V !tblIn:symtab ^tblOut:symtab
-> ID ^idn ":" T ^type ";"

...

end MicroModula.

```

5.7 作用域与标识符类别

迄今为止，登记到符号表中的标识符只有局部变量名；而在真正的程序设计语言中，还有过程名和函数名、类型、常量等，每一类标识符处理起来会略有不同。此外，在 **Pascal**、**Modula-2** 这类块结构语言中，如果相同的标识符并不是声明在同一作用域层次，则该标识符在符号表中不必是惟一的。为同时考虑上述两个方面的改进，我们只需为 **Micro-Modula** 语言添加一类新的标识符，即无参数的函数。

5.7.1 标识符作用域的文法

为给文法引入新的特性，我们添加了两个新的非终结符 **A** 和 **H**，并修改了另外三条产生式 (**M**、**B** 和 **F**)，如代码清单 5.4 所示。新的非终结符 **H** 定义了一个函数过程的语法，具有一个名字和类型、自己的局部变量、可能嵌套的函数声明以及一个语句体。**RETURN** 语句也加入到语句的定义中，这在一个块的结尾处是必不可少的。最后，修改了因子 **F** 的产生式，从而在语法上区分了函数调用和变量引用，这里采用与 **Modula-2** 语言一致的语法形式。这一点很重要，使得我们可以从语法上确定将标识符编译为一个变量引用还是一个函数调用。第 8 章介绍了如何处理同名异义的编译问题，也就是在语法上相同但其语义不同的名字，例如 **Pascal** 语言中一个变量的引用与一个无参数函数的调用。

代码清单 5.4 修改 **Micro-Modula** 语言的语法以添加函数

```

A
-> "(" ")" { 一个函数调用 }
-> ε { 一个变量引用 }

H
-> "PROCEDURE" ID ":" T ";"
    "VAR"
    V
    H
    "BEGIN"
    B
    "END" ID ";"
    H
-> ε ;

M
-> "MODULE" ID ";"
    "VAR"
    V

```

```

      H
      "BEGIN"
      B
      "END" ID "." ;

B
-> ID "!=" E ";"
  B
-> "RETURN" E
-> ε ;

F
-> "(" E ")"
-> ID A
-> NUM
-> "TRUE"
-> "FALSE" ;
```

为支持新的特性，符号表也变得更加复杂。符号表中必须引入词法层次这一概念，作为标识符的属性之一；如果同一标识符的每次出现具有不同的词法层次，则该标识符在符号表中可能出现多次。当前词法层次将作为符号表本身的一个属性来维护，因而不必由属性求值运算符 **into** 定义，但需要另外一个属性求值函数 **open** 通过提升符号表当前的词法层次创建一个新的作用域。由于 **open** 总是返回一个新的符号表，只要简单地不再引用更高层次派生出的符号表，就可以降低词法层次（从而更高层次符号表中的所有符号被自动丢弃）。

此时还有必要区别符号表中的两类标识符，即变量和函数。它们有相同的类型值，但变量名不可用于函数调用，而函数名不可出现在赋值语句的左边，也不可无空参数表括号就出现在表达式中。之前我们将 **INTEGER** 和 **BOOLEAN** 这两个类型分别编码为简单的数值 1 和 2，现在还须包括其类别。将符号表中的每一个值改为一个同时包含类型字段和类别字段的记录，即可解决这一问题。**TAG** 编译程序具有一种为记录进行编码的机制，但其中的复杂性会令读者分散注意力；因而我们手工将两个字段分别编码为十进制整数的十位数和个位数，如表 5-1 所示。一个登记在符号表中的标识符现在通过自己的值携带了两段信息：类别和类型。当检索到一个值时，必须从中抽取信息以还原各部分的含义；采用普通的整数乘法和除法执行压缩和解压功能，即可达到上述目的。

表 5-1 将类别和类型压缩为单个整数：例如，整数变量为 31、布尔函数为 42

Kind (类别)	3	变量
	4	函数
Type (类型)	1	整数类型
	2	布尔类型
压缩:	$\text{packedvalue} = \text{kind} * 10 + \text{type}$	
解压:	$\text{kind} = \text{packedvalue} / 10$	
	$\text{type} = \text{packedvalue} - \text{kind} * 10$	

约束检查除了验证标识符的用法以及新语法中显而易见的语义之外，还需要两个新的属性求值以实现 **Micro-Modula** 语言的函数，其中一个涉及符号表的管理。在每一函数名的声明之

后、该函数声明的入口处，会创建一个新的作用域，从而在作用域关闭后函数名仍然可用。创建新作用域还有另一好处，就是在新作用域中声明的符号会在作用域关闭时自动丢弃，而这正是块结构语言的要求。

另一个新的属性断言稍微模糊一些，但类似于要求验证 **RETURN** 语句中表达式的类型必须与函数的类型相匹配，以及 **RETURN** 语句未被省略。由于上述文法要求 **RETURN** 语句是最后一条语句，原本可将该语句加到函数头部 **H** 的产生式中，从而在语法上作出硬性规定；但这里采用的方法更加通用，并且适用于 **RETURN** 语句的出现位置未作如此限制的更大型语言。这里采用的方法是为表示函数体的非终结符 **B** 添加一个关于类型的继承属性，该属性将函数的类型信息向下携带给函数体，在函数体中可与 **RETURN** 表达式的类型进行比较，若无 **RETURN** 表达式则断言其值为 0。

5.7.2 标识符作用域例子分析

若不考虑因嵌套的函数声明而引发的变量作用域问题，符号表属性求值函数显然已正确地实现了 Micro-Modula 语言的类型检查。代码清单 5.5 中的文法还不够明显；考虑图 5-5 所示的一个小程序并追踪其属性流，其中特别注意符号表的变化。带编号的圆圈指出了我们对符号表和其他属性感兴趣的一些快照点。

代码清单 5.5 带函数的 Micro-Modula 语言的属性文法

M

```
->  "MODULE" ID ^idn ";"
    "VAR"
      V !<> ^vartable
    H !vartable ^bodyTable
    "BEGIN"
      B !bodyTable !0
    "END" ID ^idn "." ;

H !tblIn:symtab ^tblOut:symtab

->  "PROCEDURE" ID ^idn ":" T ^type ";"
    [into !tblIn !idn !type+40 ^nextable]
    [open !nextable ^nutbl]
    "VAR"
      V !nutbl ^vartable
    H !vartable ^bodyTable
    "BEGIN"
      B !bodyTable !type
    "END" ID ^idn ";"
    H !nextable ^tblOut

->  [tblOut = tblIn] ;

V !tblIn:symtab ^tblOut:symtab

->  ID ^idn ":" T ^type ";"
    [into !tblIn !idn !type+30 ^nutbl]
    V !nutbl ^tblOut
```



```
-> [tblOut = tblIn] ;

T ^type:int

-> "INTEGER" [type = 1]

-> "BOOLEAN" [type = 2] ;

B !tblIn:symtab !typeR:int

-> ID ^idn "!=" E !tblIn ^type ";"
    [from !tblIn !idn ^typeID]
    [typeID / 10 = 3; type = typeID - 30]
    B !tblIn !typeR

-> "RETURN" E !tblIn ^typeR ";"

-> [typeR = 0] ;

E !tblIn:symtab ^type:int

-> S !tblIn ^stype C !tblIn !stype ^type ;

C !tblIn:symtab !typeIn:int ^typeOut:int

-> "=" S !tblIn ^ctype [typeIn = ctype; typeOut = 2]

-> [typeOut = typeIn] ;

S !tblIn:symtab ^type:int

-> F !tblIn ^type P !tblIn !type ;

P !tblIn:symtab !type:int

-> "*" F !tblIn ^type P !tblIn !type [type = 1]

-> "AND" F !tblIn ^type P !tblIn !type [type = 2]

-> ;

A !typeID:int ^type:int

-> "(" ")" [typeID / 10 = 4; type = typeID - 40]

-> [typeID / 10 = 3; type = typeID - 30] ;

F !tblIn:symtab ^type:int

-> "(" E !tblIn ^type ")"

-> ID ^idn [from !tblIn !idn ^typeID]
    A !typeID ^type

-> NUM ^value [type = 1]
```

-> "TRUE" [type = 2]
-> "FALSE" [type = 2] ;

```
MODULE NestedFunctions;  
VAR  
  a: INTEGER;  
  b: BOOLEAN;  
  c: INTEGER; ①  
  
PROCEDURE First: INTEGER;  
VAR  
  a: BOOLEAN; ②  
  
  PROCEDURE Second: INTEGER;  
  VAR  
    a: INTEGER;  
    b: INTEGER;  
  BEGIN  
    b := c;  
    a := First() * Second(); ③  
    RETURN a * b  
  END Second;  
  
  BEGIN  
    a := First() = Second(); ④  
    RETURN 5  
  END First;  
  ⑤  
  
  PROCEDURE Third: INTEGER;  
  VAR  
    c: BOOLEAN;  
  BEGIN  
    c := (First() = a) = b; ⑥  
    RETURN c  
  END Third;  
  ⑦  
  
  BEGIN (* NestedFunctions *)  
    a := 1;  
    c := 2;  
    b := (First() = 3) = Third()  
  END NestedFunctions.
```

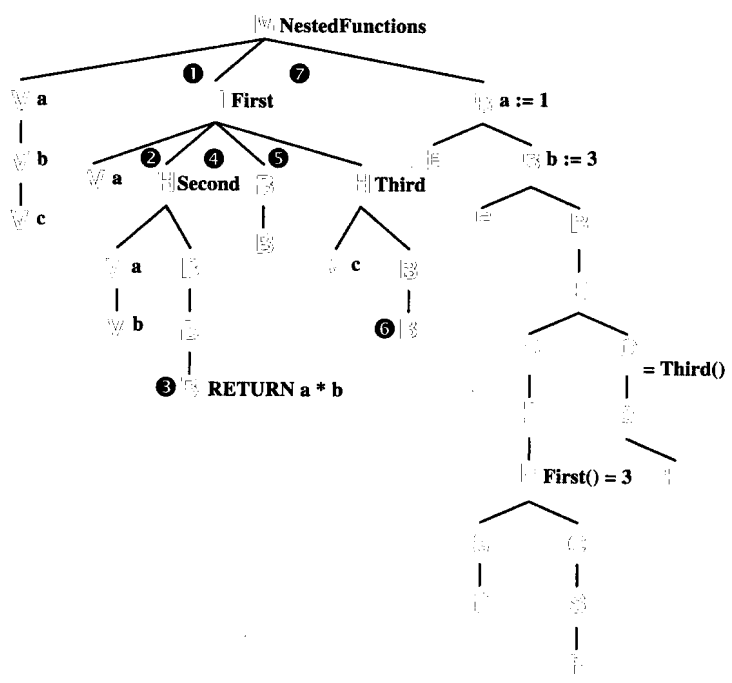


图 5-5 一个 Micro-Modula 程序与分析树，其中展示了属性流

在表 5-2 所示的快照点 1 中，符号表包含在非终结符 **M** 的产生式的一个名为 **varitable** 的属性中；该符号表恰好包含三个全局变量 **a**、**b** 和 **c**，均由递归的非终结符 **v** 推导出来，并将由表示函数头部的非终结符 **h** 继承。非终结符 **h** 的第一条非空产生式将函数名填入符号表(推导出 **nextable**)，然后为此创建一个新作用域，并将结果向下传送给 **v**。非终结符 **v** 声明了一个变量 **a** 并与符号表（也称为 **varitable**）一起返回，这正是快照 2 的基础。

表 5-2 Micro-Modula 程序中的属性流（快照 1~2）

①			
符号	词法层次	值	当前词法层次: 0
=====			
a	0	31	
b	0	32	
c	0	31	

(续)

②

符号	词法层次	值	当前词法层次: 1
a	0	31	
b	0	32	
c	0	31	
First	0	41	
a	1	32	

再回到 **H**，它递归地进入 **H**，将第二个函数的名字 **Second** 及其类型填入符号表（位于词法层次 1，因为它嵌套在函数 **First** 中），然后在词法层次 2 创建另一新的作用域，并声明了局部变量 **a** 和 **b**。这两个变量均已出现在符号表中（参阅表 5-3 的快照 3），但位于更低的词法层次（分别为 1 和 0），因而添加这两个新的标识符不会产生任何问题。

表 5-3 Micro-Modula 程序中的属性流（快照 3~7）

③

符号	词法层次	值	当前词法层次: 2
a	0	31	
b	0	32	
c	0	31	
First	0	41	
a	1	32	
Second	1	41	
a	2	31	
b	2	31	

④

符号	词法层次	值	当前词法层次: 1
a	0	31	
b	0	32	
c	0	31	
First	0	41	
a	1	32	
Second	1	41	

⑤

符号	词法层次	值	当前词法层次: 0
a	0	31	
b	0	32	
c	0	31	
First	0	41	

⑥

符号	词法层次	值	当前词法层次: 1
a	0	31	

(续)

	b	0	32	
	c	0	31	
	First	0	41	
	Third	0	41	
	c	1	32	
⑦	符号	词法层次	值	当前词法层次: 0
	a	0	31	
	b	0	32	
	c	0	31	
	First	0	41	
	Third	0	41	

快照 3 也反映了符号表经过函数 **Second** 的整个函数体后的状态。因而，分析程序在第一条赋值语句中遇到对变量 **b** 的引用时，将查找符号表（首先找到最近出现的符号），并且整数类型的局部变量 **b** 将先于布尔类型的全局变量 **b** 被找到。另一方面，在这条赋值语句的表达式部分，因子 **F** 将找到全局变量 **c**（由于不存在该名字的局部或中间标识符）。

当非终结符 **H**（分析函数 **Second**）完成其任务后，它将只新加了该函数名的符号表（**H** 中的属性 **nextable**）向下传递给其右边的函数声明；后者（由于为空）将原封不动地作为 **tblOut** 返回，从而原样传回给之前对非终结符 **H** 的调用（仍在分析函数 **First**）。符号表原封不动地沿函数体向下传递，如表 5-3 中的快照 4 所示。注意，此处所谓“原封不动”，是指分析树中每一快照点表示的符号表属性的所有副本具有相同的值。

快照 5 再次展示了非终结符 **H** 分析完 **First** 函数体后传出的符号表属性内容，它们将向下传递给递归调用的非终结符 **H** 以分析函数 **Third**。标识符 **Second** 及其所有局部变量以及 **First** 的所有局部变量全部消失——实际上它们从未在这一份符号表中出现过。因而，这是与 **Modula-2** 和 **Pascal** 语言的作用域可见性规则相一致的，函数 **Second** 在函数 **Third** 中是不可见的（参阅快照 6），它在主程序体中也是不可见的（如快照 7 所示）。

5.7.3 符号表的其他问题

上述例子仅展示了无参数的过程；将参数引入类型系统后，情况会变得更加复杂。值参在局部处理，与局部变量相似，惟一区别是它们在过程调用中被赋予了一个初始值。参数的个数和类型也反映了一个过程的功能接口，**Modula-2** 语言将其视为一个过程类型的组成部分。强类型检查禁止以错误的参数个数或类型调用一个过程。由于我们采用的符号表编码机制缺乏足够的空间在单个过程类型的记录中表达任意数目的参数类型，故将此问题留作下一章的练习。

变参（又称引用参数，**Modula-2** 语言在参数表中用关键字 **VAR** 标识）引入了一些新的问题，许多现代程序设计语言（例如 **C** 语言）甚至不支持变参。变参的问题之一是必须按不同方式分析这两类参数的过程调用语法：值参可接受类型正确的任一表达式，而变参则只能传递一个变量的引用。由于本章的重点是语法制导的语义，变参问题将推迟到第 8 章讨论。

每一程序设计语言均包含了一个由预定义函数与过程组成的标准库。其中有许多函数或过程往往带有神奇的性质，需要由编译程序进行特殊的处理。其他的库例程则与用户自己编写的

过程没有区别（从编译程序的角度看），只需将它们的声明添加到默认的（空的）符号表即可，添加时让这些声明出现在主程序直接外层的词法层次。那些神奇的过程则通常需要在属性文法中有一些特殊的产生式来处理其特定的属性。例如，Modula-2 语言包括几个多态的库过程，即这些过程可接受某一类数据类型中的任一类型的参数。其中之一是过程 **INCL**，它的第一个参数接受一个集合类型的任意变量，第二个参数必须是一个与集合基类型兼容的普通表达式的值；因而，如果集合是 **SET OF CHAR**，则第二个参数的值必须是 **CHAR** 类型，如此类推。Modula-2 语言不允许用户编写一个多态的过程，因而编译程序必须包含特殊的约束以检查传递给一个多态库例程的参数。Ada 语言的程序员可编写多态的过程，但 Ada 语言的另一一些库例程也需要特殊的约束程序进行处理。在一个典型编译程序中，与语言中所有其他的约束一样，我们也可花同样的力气定义这些库的产生式。

Modula-2 语言采用与其数据封装机制 **MODULE** 类似的语法，为强类型语言引入单独编译的特性。内部模块的实现不会带来什么惊奇，仅要求在模块边界创建一个新的（空）符号表。封闭的环境必须以一种对查表操作不透明的方式与新的符号表之间建立链接，但是对 **IMPORT** 和 **EXPORT** 子句的语义处理必须能够访问它。单独编译的定义模块和实现模块意味着在这些编译单元之外有一虚拟的环境，该环境通常实现为符号定义文件和目标代码文件。这里仅提供一些启示，并不深入讨论与处理模块声明相关的特殊问题，勤奋的学生可在一道有趣且有挑战性的练习中遇到这类问题。

5.8 在递归下降中实现属性

第 4 章展示了从一个 LL(1) 上下文无关文法到 Modula-2 这类程序设计语言的递归下降分析程序之间的一一映射。综合属性和从左到右的继承属性及其求值函数可直接添加到递归下降分析程序中，就好像添加到构造分析程序时所参照的文法中那样简单。

每一继承属性均作为实现一个非终结符引用的过程调用中的一个值参传递；每一综合属性则作为过程调用中的一个变参传递。每一属性求值函数是分析程序代码中的一条赋值语句，将计算得到的值赋值给一个指定的属性；每一属性谓词则实现为条件测试，其选项为 **Error**。所有不在参数表中的综合属性均实现为分析程序代码中的局部变量。

例如，考虑 Micro-Modula 语言的属性文法中的非终结符 **C**：

```
C ↓tblIn:symtab ↓typeIn:int ↑typeOut:int
  → "=" S ↓tblIn ↑ctype [typeIn = ctype; typeOut = 2]
  → [typeOut = typeIn] ;
```

实现该产生式的 Modula-2 语言代码显而易见，如代码清单 5.6 所示。Micro-Modula 语言实现的剩余部分留作读者练习。

代码清单 5.6 属性文法中一条产生式的 Modula-2 语言实现

```
PROCEDURE C(tblIn: symtab; typeIn: INTEGER; VAR typeOut: INTEGER);
VAR
  ctype: INTEGER;
BEGIN
  IF Nexttoken = '=' THEN
    Getoken;
    S(tblIn, ctype);
    IF typeIn <> ctype THEN
```

```
        Error
    END;
    typeOut := 2
ELSE
    typeOut := typeIn
END
END C;
```

5.9 实现符号表

正如这两者所表现的功能所示,符号表不会、也不应与字符串表相同。字符串表管理由字符组成的串,从中抽取简单的序数句柄以便于后续的引用。为避免重复浪费,每一新标识符或字符串常量必须与表中已有的串逐个字符地进行比较;这也同时执行了相等匹配,而相等匹配在符号表中又是如此重要。字符串表中仅存放了字符串本身的拼写以及高效管理所需的结构,除此之外别无他物。

符号表是一种数据结构,其中存放了与特定标识符相关联的若干数据。在一种块结构语言中,标识符可在符号表中重复出现并关联到不同的数据。

然而,符号表与字符串表两者之间还是有些渊源的。在历史上,字符串表往往嵌入在符号表中。这意味着并非将字符串抽取到一个单独的表中,而是将标识符的整个字符串存储在符号表中,因而实际上符号表的查找是基于标识符拼写的字符串比较。为避免浪费多余的时间,标识符被限制只能含有较少数目的字符(例如6个或8个字符)。

另一渊源是这两张表都有一个功能是查找一个标识符;实际上,字符串表的存在就是为了简化符号表中的查找功能。字符串表仅当遇到一个新的标识符时才会增大,而符号表则可能随着程序的块结构而增大或缩小。字符串表仅向其数据结构查询某一特定的项目是否存在,而符号表则不仅查询是否存在,还查询该标识符所存储的值记录是什么。

由于块结构语言的符号表中标识符可能重复出现,采用第3章介绍的字符串表查找优化的同类技术未必总是实用的。主要困难在于当每一过程的作用域关闭时,当前级别的所有符号必须从表中删除。除了维护一个简单的栈结构线性表之外,还另有几种办法可实现这一需求。

类似于字符串表,符号表的最直接查找优化仍是散列表。然而,为保持作用域嵌套关系,符号表的每一词法层次必须有自己的散列表。有两种策略均可提高效率,取决于表的相对大小以及向上层查找的次数。一种很简单的做法是只要创建了一个新的词法层次,就将上一层的整个散列表复制到其中。新添加的标识符建立在旧表的基础上,重声明的标识符直接取代表中原有的声明。这种仅查找一张表的做法可取得最佳的查找效率,不管待查找的标识符位于作用域链上的多高层次;不足之处是整张散列表及其所有的桶在每次创建新作用域时都必须重新复制。这在概念上不成问题,因为我们将属性求值也理解为复制,但这会花费一些时间;幸运的是这只会发生在模块、过程和记录的边界处发生。

另一种做法是在每一词法层次创建一个新的散列表,并链接到列表中的下一层。如果在当前作用域的散列表找不到某一符号时,则沿着链接依次查找每一散列表,直至找到该符号。如果在众多词法层次中每一层次仅声明了少量标识符,则这一做法退化为与线性栈相同的性能。然而,程序的统计研究表明,大多数符号引用集中在列表中的局部和全局两端。若在局部表中查找一个符号失败,则立即查找全局散列表,然后仅当查不到时才到直接链接的表中查找该符

号,这样可大大提高性能。只要在某局部作用域中重声明了一个全局符号,在全局表中即设置一个标记以消除这一查找捷径。重声明全局标识符往往很少见,故可永久设置这些标记且不会严重影响效率。

符号表的另一结构取决于字符串表管理程序的支持。如果标识符被顺序地编号(在扫描程序中是很简单的步骤),符号表可组织为记录指针的线性向量(数组),这些指针可通过标识符编号直接作为下标。这样不再需要任何查找,因为所有标识符的信息记录都是一次访问;每当创建一个新作用域,指针数组被复制到一个新的向量。从性能角度看,该方法相当于一个完美散列表,在散列桶中不存在任何冲突,即每一散列桶中最多只有一个元素。然而,符号表必须与标识符的最大数目一样长。

小结

属性文法为程序设计语言的语义规格说明提供了一种方便的工具,也为实现编译程序和编译程序编写系统提供了形式化基础。借助于属性文法已开发了多种编译程序生成工具:MUG1和MUG2(Modularer Übersetzer Generator,均为完整的编译程序生成工具)、GAG-A(基于属性文法的生成工具)、HLP 84(Helsinki Language Processor 84,是一个语言处理工具箱)和TAG(本书中介绍),以及P. Deransart在1988年列举的另外33种现有的编译程序生成工具系统。属性文法不仅可用于编译程序的自动生成,实践表明它还有助于生成文本编辑环境(参阅[Horowitz & Teitelbaum, 1986])以及程序优化(本书第8章将全面讨论的课题)。

一个属性文法由一个上下文无关文法和一个附加到文法每一产生式的语义规则集组成。与每一非终结符相关联的0个或多个属性的值由这些语义规则定义或约束。一个输入串(属于某一语言)的含义由属性文法中目标符号的属性值确定。有两类属性:综合属性和继承属性。综合属性沿分析树向上传递信息,继承属性沿分析树向下传递信息;将综合属性和继承属性组合在一起,可在一棵树的兄弟结点之间传递信息。

本章采用Micro-Modula语言的文法展示了这一技术,描述了如何在一个属性文法中检查程序设计语言的特定约束。本章介绍的类型检查方法是在符号表中存储标识符的相关信息。最后,本章扩展了第4章介绍的将一个LL(1)上下文无关文法映射为一个递归下降分析程序的方法,说明了如何将综合属性和继承属性添加到一个递归下降分析程序中;继承属性作为过程调用中的值参传递,综合属性则作为变参传递;属性谓词实现为条件测试。

符号

↓ 继承属性,记为↓attname。

↑ 综合(派生)属性,记为↑attname。

关键术语

assertion (断言) 在一条产生式的上下文中约束求值结果为真的语句或表达式。如果任一断言求值为假,则正在分析的串不属于该语言。我们区分两类断言:属性求值函数和谓词。

attribute evaluation function (属性求值函数) 约束其属性为某一独立已知的单个值,例如 $a_1 = 10$ 。

predicate (谓词) 为属性求值函数已约束的一个或多个属性添加额外限制。谓词可约束一个属性为单个值,亦可将该属性关联到一个表达式(可能涉及另外的属性),例如 $a_1 > a_2$ 。

attribute (属性) 描述一个对象的性质或特征。在属性文法中,对象是指一个非终结符或它代表的子串。上下文无关文法中的每一非终结符可与一个属性集相关联,这些属性描述了该非终结符所代表的对

象的特征或性质。

evaluation order (求值次序)

- (1) 自底向上, 当属性从正考虑的子树的固有性质推导其属性值时。
- (2) 正顶向下, 当属性依赖于一棵子树所处的上下文的某一部分时。
- (3) 从左到右, 当信息由分析树中某一结点的左子树推导出, 然后由右子树继承时。
- (4) 从右到左, 当信息由分析树中某一结点的右子树推导出, 然后由左子树继承时。

inherited (继承属性)

- (1) 在产生式中一个由断言(属性求值函数)定义的属性, 这些产生式的右部引用了该属性所属的非终结符。
- (2) 记为 $\downarrow\text{attname}$ 。

synthesized (综合属性)

- (1) 一个在所属的非终结符的产生式内部定义的属性, 或所属的终结符本身固有的属性。
- (2) 亦称派生(derived)属性。
- (3) 记为 $\uparrow\text{attname}$ 。

attribute grammar (属性文法) 是一个三元组 (G, V, F) ; 其中, G 是一个上下文无关文法, V 是一个特定属性的有穷集, F 是一个属性断言的有穷集。

constrainer (约束程序) 是编译程序的一部分, 负责校验正被编译的程序的所有断言是否成立。

Micro-Modula (Micro-Modula 语言) 是 Modula-2 语言的一个精简子集, 用于展示属性文法的用法。

symbol table (符号表) 是一种将一个标识符集与一个值集相关联的数据结构。对符号表的访问可能是:

- (1) 如果标识符不在符号表中, 则添加一个新的标识符及其值。
- (2) 查找一个标识符, 并检索其值。

练习

1. 指出以下每一属性文法的总体属性值流向是自底向上、自顶向下、从左到右、从右到左、存在循环、还是其他形式。

- (a) $G \rightarrow A \downarrow 1$
 $A \downarrow n \rightarrow B \downarrow 3n \quad A \downarrow 7n$
 $\rightarrow "c" \quad C \downarrow n-1$
 $B \downarrow n \rightarrow "a" \quad B \downarrow n+4 \quad "b" \quad C \downarrow 2n$
 $\rightarrow "b"$
 $C \downarrow n \rightarrow "c"$
- (b) $G \rightarrow A \uparrow x$
 $A \uparrow x \rightarrow B \uparrow u \uparrow v \quad A \uparrow y \quad [x = u y + v]$
 $\rightarrow "c" \quad C \uparrow z \quad [x = 2 z]$
 $B \uparrow u \uparrow v \rightarrow "a" \quad B \uparrow r \uparrow s \quad "b" \quad C \uparrow x \quad [u = 2 r + x - s; \quad v = s + 1]$
 $\rightarrow "b" \quad [u = 1; \quad v = 2]$
 $C \uparrow x \rightarrow "c" \quad [x = 3]$
- (c) $G \rightarrow A \downarrow 0 \uparrow r$
 $A \downarrow x \uparrow z \rightarrow B \downarrow y \uparrow z \quad A \downarrow x \uparrow y$
 $\rightarrow "c" \quad C \downarrow x \uparrow y \quad [z = 10 y + 3]$
 $B \downarrow x \uparrow w \rightarrow "a" \quad B \downarrow 10y+2 \uparrow z \quad "b" \quad C \downarrow x \uparrow y \quad [w = 10 z + 1]$
 $\rightarrow "b" \quad [w = 10 x + 2]$
 $C \downarrow x \uparrow y \rightarrow "c" \quad [y = 10 x + 3]$

(d) (低效的内存分配程序)

$$\begin{aligned}
 D &\rightarrow V \downarrow 0 \uparrow m \\
 V \downarrow a \uparrow m &\rightarrow N \downarrow a \quad T \downarrow a+k \uparrow n \uparrow k \quad V \downarrow n \uparrow m \\
 &\rightarrow \varepsilon & [m=a] \\
 T \downarrow a \uparrow x \uparrow k &\rightarrow N \downarrow a \quad T \downarrow a+k \uparrow x \uparrow k \\
 &\rightarrow "b" & [x=a; \quad k=1] \\
 &\rightarrow "i" & [x=a; \quad k=2] \\
 &\rightarrow "r" & [x=a; \quad k=4] \\
 N \downarrow a &\rightarrow "s"
 \end{aligned}$$

(e) (改进的内存分配程序)

$$\begin{aligned}
 D &\rightarrow V \downarrow 0 \uparrow m \\
 V \downarrow a \uparrow m &\rightarrow "s" \quad T \downarrow a \uparrow n \uparrow k \quad N \downarrow n \quad V \downarrow n+k \uparrow m \\
 &\rightarrow \varepsilon & [m=a] \\
 T \downarrow a \uparrow x \uparrow k &\rightarrow "s" \quad T \downarrow a \uparrow y \uparrow k \quad N \downarrow y \\
 &\rightarrow "b" & [x=y+k] \\
 &\rightarrow "i" & [x=a; \quad k=1] \\
 &\rightarrow "r" & [x=a; \quad k=2] \\
 &\rightarrow "r" & [x=a; \quad k=4] \\
 N \downarrow a &\rightarrow \varepsilon
 \end{aligned}$$
(f) $G \rightarrow E \uparrow r$

$$\begin{aligned}
 E \uparrow x &\rightarrow E \uparrow v \quad "+" \quad T \downarrow v \uparrow f & [x=v+f] \\
 &\rightarrow T \downarrow 0 \uparrow x \\
 T \downarrow n \uparrow m &\rightarrow T \downarrow x \uparrow m \quad "*" \quad F \downarrow n+m \uparrow x \\
 &\rightarrow F \downarrow n \uparrow m \\
 F \downarrow x \uparrow y &\rightarrow F \downarrow v \uparrow y \quad D \uparrow v \\
 &\rightarrow D \uparrow v & [y=x+v] \\
 D \uparrow v &\rightarrow "0" & [v=0] \\
 &\rightarrow "1" & [v=1]
 \end{aligned}$$
(g) $G \rightarrow E \downarrow 0 \uparrow r$

$$\begin{aligned}
 E \downarrow s \uparrow r &\rightarrow E \downarrow a+s \uparrow r \quad "+" \quad T \downarrow 1 \uparrow a \\
 &\rightarrow T \downarrow 1 \uparrow a & [r=a+s] \\
 T \downarrow m \uparrow v &\rightarrow T \downarrow m \uparrow p \quad "*" \quad F \downarrow p \downarrow 1 \uparrow v \\
 &\rightarrow F \downarrow m \downarrow 1 \uparrow v \\
 F \downarrow p \downarrow s \uparrow v &\rightarrow F \downarrow p \downarrow 2*s \uparrow n \quad D \downarrow s \uparrow b & [v=n+b*p] \\
 &\rightarrow D \downarrow s*p \uparrow v \\
 D \downarrow s \uparrow v &\rightarrow "0" & [v=0] \\
 &\rightarrow "1" & [v=s]
 \end{aligned}$$

2. 对练习 1 中的每一属性文法, 分别为以下指定的串构造分析树; 若可能, 为每一非终结符结点注释其属性值。

(a) $abbcabbccc$ (b) $abbccc$ (c) $aabbcbcabbbccc$ (d) $ssissbsrsi$ (e) $srsissbssrsr$ (f) $011*1+1$

(g) $1101+11*10$

3. 转换练习 1 的属性文法 (c) 和 (g), 对语言中所有的串产生与原来相同的综合属性 r , 但文法中仅使用综合属性。

复习小测验

指出下列陈述是否正确。

1. 对于一个给定的属性文法, 某一特定非终结符所关联的属性集是继承属性和综合属性集合的并集。
2. 一个属性文法实际上就是一个上下文敏感文法。
3. 在一个属性文法中, 每一非终结符有一个或多个属性与之相关联。
4. 编译程序中的约束程序将校验对应文法中的每一非终结符至少有一个综合属性。
5. 一条产生式中属性 x 和 y 相关联的一个形如 $x < y$ 的断言就是所谓谓词的一个例子。
6. 谓词不会将一个属性约束为某一特定的值。
7. 在一个属性文法中, 属性求值函数无法汇集到空产生式中。
8. 同名异义是指语法相同而语义形式不同。

编译程序实验项目

1. (a) 修改你的 Itty Bitty Modula 语言文法, 在本小题中删除其中的过程和函数, 但留下 **INTEGER** 和 **BOOLEAN** 作为内置类型。为该文法添加必要的属性以及属性求值函数, 从而文法可正确地强制规定标准 Modula-2 的类型规则。
(b) 在 TAG 编译程序中编译你的文法以测试该文法, 或编写一个递归下降分析程序实现它。你的编译程序应接受正确的程序片段, 例如:

```
TYPE a = BOOLEAN;
VAR b: a; ...
IF b THEN ...
b := (3 > 2) AND (TRUE > FALSE);
...
```

但应拒绝错误的源代码文本, 将它们看作不属于该语言的串。例如:

```
TYPE a = BOOLEAN;
VAR a: INTEGER;           (* 重复声明 *)
IF a = TRUE THEN ...     (* a 被声明为一个类型 *)
b := 3 > TRUE;            (* b 未声明; 表达式中类型不兼容 *)
IF 3 THEN ...            (* 不合法的条件类型 *)
```

2. (a) 为你的文法添加无参数的函数, 其语义支持对上层变量的引用。
(b) 测试你的新文法; 保证引用了作用域之外的变量会报告一个错误, 而重声明的标识符会正确地屏蔽非局部声明。
3. 设计一个库文件的格式, 然后为你的编译程序引入单独编译的特性, 包括 **DEFINITION MODULE**、**IMPLEMENTATION MODULE** 和 **IMPORT** 子句。假设在 **DEFINITION MODULE** 中声明的所有符号会自动地导出到库中; 目前还不必尝试 **EXPORT** 子句。

进一步阅读

Deransart, P., Jourdan, M., & Lorho, B. Attribute Grammars: Definitions, Systems and Bibliography, Lecture Notes in Computer Science 323. New York: Springer-Verlag, 1988.

参阅第 I 部分的第 1、2 小节, 其中给出了属性文法的基本定义和性质; 还要特别参阅第 II

部分, 该部分评述了现有的基于属性文法的编译程序生成工具 (共 40 种)。

Horowitz, S. & Teitelbaum, T. "Generating Editing Environment Based on Relations and Attributes." ACM Transactions on Programming Languages and Systems, Vol.8, No.4 (October 1986), pp.577-608.

参阅第 2.1 小节的属性文法综述, 以及第 4.2 和 4.3 小节的关系属性文法。

Jazayeri, M. & Pozefsky, D. "Space-Efficient Storage Management in an Attribute Grammar Evaluator." ACM Transactions on Programming Languages and Systems, Vol.3, No.4 (October 1981), pp.388-404.

参阅第 396~397 页第 1 小节关于属性文法的优秀综述, 以及为属性计算栈中偏移量的伪码。

Katayama, T. "Translation of Attribute Grammars into Procedures." ACM Transactions on Programming Languages and Systems, Vol.6, No.3 (July 1984), pp.345-369.

参阅第 3 小节, 其中说明了过程调用参数与属性相关联的翻译方法。

Kennedy, K. & Ramanathan, J. "A Deterministic Attribute Grammar Evaluator Based on Dynamic Sequencing." ACM Transactions on Programming Languages and Systems, Vol.1, No.1 (July 1979), pp.142-160.

给出一种确定的方法对“语义”分析树中的所有属性进行求值; 请特别关注第 4 小节。

Kleene, S.C. Mathematical Logic, New York: Wiley, 1967.

参阅第 2 章关于谓词 (又称谓词函数) 的讨论。

Knuth, D.E. "Semantics of Context-Free Languages." Mathematical Systems Theory Journal, Vol.2 (1968), pp.127-145.

Knuth, D.E. "Semantics of Context-Free Languages: Correction." Mathematical Systems Theory Journal, Vol.5 (1971), pp.95.

Robinson, J.A. Logic: Form and Function, New York: Elsevier North Holland, 1979.

参阅第 6 章关于断言以及逻辑结论的思路的讨论。

第 6 章 语法制导代码生成

本章旨在：

- 基于代码生成序列将源代码文本翻译为目标代码，这些代码生成序列定义在一个指定源代码文本语法的文法中
- 通过一个小型虚拟计算机的指令集，熟悉目标计算机机器语言的工作方式
- 扩展第 5 章的 **Micro-Modula** 语言文法，在编译 **Micro-Modula** 程序时生成可执行的目标代码
- 利用回填技术解决向前分支问题，从而有可能实现一个“一遍”编译程序
- 揭示不同控制结构的代码生成
- 研究访问记录、数组、指针等数据结构的代码

6.1 简介

编译程序的总体目标是将源程序翻译为目标机器语言，又称目标代码。目标代码必须保持与源程序相同的语义，亦即它们必须计算出相同的结果，尽管两者在语法上有很大差异。迄今为止，本书前 5 章仅关注源程序的语法正确性，包括那些也可被称为语法的各种形式（假如我们使用了上下文敏感文法作为描述工具）；例如，“先声明、后使用”规则，以及类型用法正确性等。这些是一个编译程序前端的关注点，编译程序前端负责识别一个语法上正确的源程序。现在我们将注意力转移到后端，编译程序的后端关注如何生成目标代码。另外有一些程序使用了编译程序前端的方法，但配合不同的后端，例如解释程序、美化打印工具（分析源程序的语法并以合适的缩进方式打印出来）。

最近有一些基于形式化方法的研究采用一个属性文法描述目标机器，然后通过一个定理证明引擎（使用人工智能技术）找出一个正确的翻译程序将源程序转换为目标代码。但是这些成果往往忽视了大量需要考虑的因素，因而仍未产生一些真正可用的实用编译程序。

所以我们改为重点关注更传统的方法，为源代码的每一语法形式（手工）选择合适的目标代码序列，这种方法称为语法制导代码生成。之所以称“语法制导”，是因为代码生成序列定义在一个指定源程序语法的文法之中；换言之，分析程序可严格地基于被分析源代码的语法，去选择合适的机器语言操作序列，除常量值以及变量与过程的机器地址之外，不以任何方式依赖于文法的语义。

6.2 计算机硬件体系结构

不同的计算机硬件体系结构会对编译程序设计人员产生不同影响。尽管形式语言设计方法长期主宰着源语言的定义，但是对硬件设计的最大推动来自市场的压力（往往由机器语言代码酷爱者驱动），硬件充分展示了其影响力。不同计算机环境中奇形怪状的指令集造成两方面影响。

首先，实现某一特定源代码结构所需的最合适指令在当前正在设计的编译程序所面向的目标机器中未必存在，因而编译程序必须生成一系列其他指令以取得同一效果。

其次,对于任一给定的源代码结构,存在多种途径取得同一效果,因而编译程序必须从中选择一种。一个优秀的编译程序可基于各种评价标准动态地作出选择,例如基于内存空间或执行速度;而一个简单的编译程序则很可能采用由编译程序设计人员预先作出的武断选择。第9章将涉猎一些与代码选择有关的复杂问题。

除了仅与高性能计算机有关的指令调度需求之外,对于代码生成而言,不同计算机体系结构的最大区别在于可用寻址模式的数量与种类。最早的计算机仅有一个地方供计算机指令取数据,即主存。为实现两个数相加,必须指定内存中的两个(或三个)位置;对每一条机器指令,必须从内存中取出数据的值,再将结果放回内存中。利用累加器可消除其中的一个内存地址,从而显著地提高性能。单个值从内存中取出(装入)后放入累加器,或从累加器存储回内存;算术运算只须指定一个地址,第二个操作数总是累加器(如图6-1所示)。进一步提高性能的途径是让两个操作数都来自通用寄存器。由于有多于一个的寄存器需指定,指令又再次需要两个地址,但在指令字中寄存器地址只占用少量宝贵的位,并且在大多数计算机中访问寄存器比访问内存更快。在众多知名的计算机中,DEC公司生产的PDP-11可能拥有最简洁的体系结构,并且接近完全的正交性,亦即任一指令均可在寄存器或内存中找到两个或其中一个操作数。

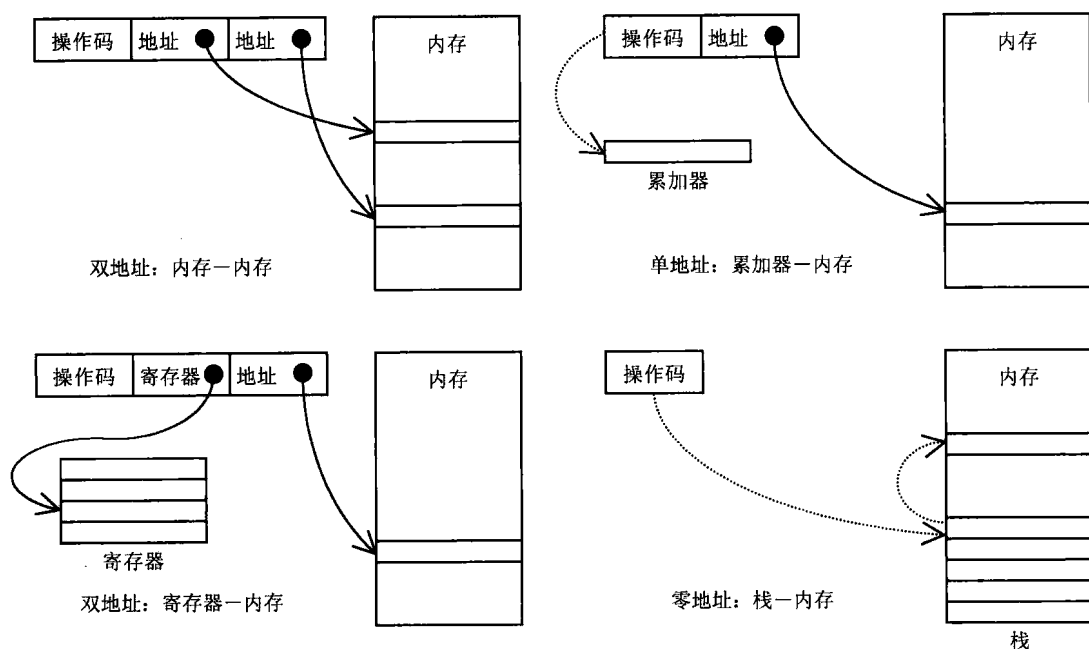


图 6-1 不同的寻址模式

随着更小型计算机的出现,更短的指令字长度变得越来越有价值,因而计算机体系结构设计人员也在寻求方法以限制指定一个操作数时所需的位数。其中的一种方法是使用寄存器;另一种方法是指派一小块内存区域(典型大小为64或256字长),从而可用更少的位数寻址;将寄存器或内存位置作为操作数的间接引用也是一种常见方法。许多小型计算机提供了多种寻址模式供选择;在一个编译程序中,选择合适且高效的寻址模式(在需要作出选择时)往往是相当困难的。

6.3 栈机器的表达式求值

如果一个地址比两个地址更好，那么可能零个地址是最好的。一个零地址计算机将其所有数据均保存在一个操作数栈的顶部。第 4 章学习了一个带栈的下推自动机 (PDA) 如何作为分析上下文无关语言的高效形式化机器，本节将展示栈体系结构也可简化表达式的求值。考虑以顺序方式对一个简单表达式 $3 * 4 + 5 * 2$ 进行求值；在从左到右的求值次序中，计算机可完成的第一个运算是子表达式 $3 * 4$ ，产生中间结果为 12。这个值不能立即与求和运算的另一个项相加，因为另一个项本身又是两个因子的积（尚未求值）。如果从右开始求值，也存在同样的问题。我们无法避免计算并保存一个中间值。

若有足够的寄存器可用，中间结果可存储在寄存器中，直至寄存器用完，但所有编译程序最终都需强制将子表达式存储到内存中的临时数据空间。一些编译程序直接将该职责移交给程序员，显示一条“请化简该表达式”之类的报错信息。在程序员卷入此项任务时，可将临时数据空间分配给一些变量；而由编译程序自动处理时，依然是分配临时数据空间，但此时的分配方式是匿名的。一种更简单的方法是由一个表达式栈提供支持。

考虑如图 6-2 所示表达式栈上的操作序列。各个值按在表达式中出现的从左到右次序压入栈中，只要一个加法或乘法运算的两个操作数是栈顶的两个元素，该运算就删除（弹出）两个操作数，再将运算结果压入栈中。

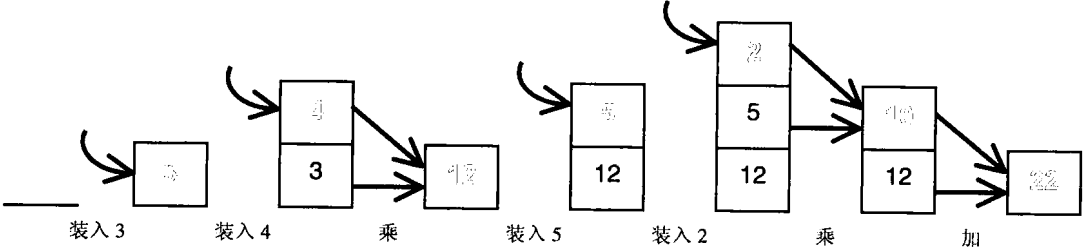


图 6-2 表达式 $3 * 4 + 5 * 2$ 的栈求值过程

波兰数学家 Lukasiewicz (卢卡西维茨) 证明了对任意带括号和运算符优先级（正如我们所熟悉的算术运算一样）的表达式，可通过将运算符置于其操作数之前，等价地表达为没有括号和优先级的形式。因而，表达式 $3 * 4 + 5 * 2$ 和 $(3 * 4) + (5 * 2)$ 均可表达为无括号的形式： $+, *, 3, 4, *, 5, 2$ 。因当时特殊的沙文主义，他的工作由一位只记得发明者的国籍而忘记其本人名字的学者发表在美国刊物上，因而被称为“波兰表示法”并沿用至今。将运算符移至操作数右边也可保持等价性，这种表示法通常称为逆波兰表示法或后序波兰表示法，但往往也只称波兰表示法。上述表达式的逆波兰表示法为 $3, 4, *, 5, 2, *, +$ 。

图 6-3 展示了同一表达式 $3 * 4 + 5 * 2$ 使用文法 G_2 构造的分析树。注意，以一种从左到右、后序遍历的方式访问树中每一结点时，这一次序恰好与这些结点附带的语义操作需执行的次序完全相同（这也正巧是分析程序构造它们时的同一次序）。这意味着当从顶部出发进入每一结点时，先向下遍历左子树、再到右子树，最后才正式访问该结点，作为退出顶部结点前的最后一件事。在这一例子中，进入分析树顶部 E 的第一条产生式后，立即下降到其下左边的结点 E，然后依次到结点 T、T 和 F；在从叶结点（单词 3）退出时“访问”F 并产生“Load 3”；到达

(续)

代 码	助记符	功 能
17	Less	弹出 a；弹出 b；若 $b < a$ 则压入 1，否则压入 0
18	Greater	弹出 a；弹出 b；若 $b > a$ 则压入 1，否则压入 0
20	Negate	弹出 a；压入 $-a$
1	BranchFalse	弹出偏移量；弹出值；若该值为 0 则将该偏移量加到程序计数器 (PC)
3	Call	弹出地址；压入返回地址，跳转到了例程
4	Enter	弹出数字；为该数字指定数量的变量分配空间
5	Exit	弹出数字；为该数字指定数量的参数回收空间，并返回调用者
8	Dupe	压入栈顶的一个副本
9	Swap	弹出两个值；以相反次序压回栈中
24	Stop	停止运行
25	Global	栈顶减去帧指针 (FP)

留意在这些指令中，仅有 **LoadCon** 指令具有显式的操作数，它只是从跟在该指令后面的内存字中将一个常量装入栈中。所有其他指令都是真正的零地址，并且从栈中取走它们的所有操作数。

如前所述，**Multiply** 和 **Add** 从栈中弹出两个操作数，执行各自的操作，然后将结果压入栈中；布尔运算符 **Or** 和 **And** 的工作方式类似。比较运算符 **Equal**、**Less** 和 **Greater** 也弹出两个操作数，按运算符名字所示含义进行比较，然后将 1 (真) 或 0 (假) 压回栈中。**Negate** 只从栈顶取走一个操作数，再将其算术求反结果压入栈中。**Load** 从栈顶弹出一个地址 (当然这只是一个数字而已)，利用该数字的值在内存中寻址得到某一位置，然后将该内存单元的内容的一个副本压入栈中。类似地，**Store** 弹出一个值和一个地址，将值存储到该地址指定的内存单元中。**Zero** 是 **LoadCon 0** 的同义写法，即它们有相同的含义 (机器上的效果) 但有不同的写法 (即具有不同的操作码)。控制结构没有那么直观，稍后将更详尽地讨论它们。

习惯上，我们将注释写在助记符及其可能拥有的所有操作数的右边，单独作为一列。由于不存在汇编程序将 **IBSM** 助记符转换为机器代码，并且由于没人会以任何手工方式书写 **IBSM** 机器指令 (除非像本章练习那样，目的是为了让你熟悉该指令集)，我们可接受以相当自由的方式表述这些指令。在本书的练习中，鼓励你无拘无束地为代码书写注释。

为执行简单表达式 $3 * 4 + 5 * 2$ 的求值，必须为栈机器生成恰好 7 条指令，且次序必须相同。据图 6-2 可得 (留意每一行注释栏所示的栈内容)：

指令助记符	产生的栈内容 (栈顶在左边)
LoadCon 3	3
LoadCon 4	4, 3
Multiply	12
LoadCon 5	5, 12
LoadCon 2	2, 5, 12
Multiply	10, 12
Add	22

如果某一操作数是一个变量而不是常量，则该变量的地址将作为一个常量装入，然后用 **Load** 指令取出该变量的值。因而，赋值语句 **a := b** 可编译为如下序列：

指令助记符	产生的栈内容（栈顶在左边）
LoadCon <a 的地址>	a 的地址
LoadCon <b 的地址>	b 的地址, a 的地址
Load	b 的值, a 的地址
Store	(空)

Store 指令要求栈中有两个值：栈顶的值是待存储的值，它由上一条 **Load** 指令（该指令则从变量 **b** 中取出该值）压入；弹出的第二个项目是变量 **a** 在内存中的地址，它由第一条 **LoadCon** 指令压入。**Store** 指令要求这两个值在栈中的次序是地址位于待存储的值的下方（之前压入）；因而，地址必须先压入，然后在存储表达式的值之前必须先计算该表达式的值。我们很快将看到，这正是我们能够产生这些值的次序。现在考虑赋值语句 $a := a - 1$ ，由于 **IBSM** 没有减法指令，我们将 **Negate** 和 **Add** 两个操作合二为一，从而表达了定义减法的代数恒等式： $a - b = a + (-b)$ 。

指令助记符	产生的栈内容（栈顶在左边）
LoadCon <a 的地址>	a 的地址
LoadCon <a 的地址>	a 的地址, a 的地址
Load	a 的值, a 的地址
LoadCon 1	1, a, a 的地址
Negate	-1, a, a 的地址
Add	a - 1, a 的地址
Store	(空)

在真实的计算机中，所执行的指令只是内存中的一些数字，每一数字都表明了它们被取出时将执行的操作。因而，要生成 **IBSM** 的代码就必须生成将所需操作编码后的数字序列。假设变量 **a** 位于数字 3 编址的内存单元中，则赋值语句 $a := a - 1$ 可编译为以下数字序列：

28	LoadCon <a 的地址>
3	
28	LoadCon <a 的地址>
3	
27	Load
28	LoadCon 1
1	
20	Negate
12	Add
26	Store

代码生成的语义可展现在文法中，与所有其他语义的处理几乎完全相同：将它们括在方括号中。**Micro-Modula** 语言（其完整的文法请参阅代码清单 5.1）中乘法的产生式（重写规则）在添加了代码生成语义后，形如：

$$P \rightarrow "*" F [" 11 "] P$$

该规则阅读起来的意思是，在分析了一条乘法产生式中紧跟乘号的因子后，编译程序应将数字 11 发送到输出代码文件中。待生成的代码与单词一样放在引号中；括住代码的方括号避免了混淆代码与单词。**TAG** 编译程序允许在方括号中用引号括起来的任意字符串作为输出代码，并原封不动地将该字符串写进输出代码文件中。因而，我们引入另外的空格分隔数字 11，以表示与它之前或之后的其他数字相乘。

一些分析程序生成工具仅允许语义动作放在产生式右部的最后；这不会造成什么问题，因为非终结符可放在任何地方，并且空产生式可用于生成代码。事实上，更简便的做法通常是将代码生成产生式汇集到文法中的某个地方，以便于阅读和编辑。因而，上述单条产生式可改写为具有相同效果的两条产生式而不破坏正确性，它们的新特点是所有语义动作都位于它们所处的产生式右部的最后：

$$P \rightarrow "*" F M P$$

$$M \rightarrow [" \text{ } 11 \text{ "}]$$

注意，如果没有语义则 M 将是一个空产生式，它不会识别任何输入单词。如果应用第 2 章介绍的从一个上下文无关文法中消除空产生式的算法，所得结果即为原文法。

通常代码生成程序不仅要在一个文本文件中生成一个数字序列，还须包括一些具有特定格式的二进制目标模块文件。本书并不打算深入讨论这些复杂的文件格式，以空产生式的形式编写适当的语义动作例程即可轻易解决这些问题。

6.5 带属性的代码生成

代码生成程序直接输出文本字符串已足以处理算术运算符这类简单的语法制导代码，然而许多输出的代码依赖于语义信息，特别是那些通常存储在符号表中的数据。事实上，在大多数现代程序设计语言中，即使是运算符也是重载的，不同类型的操作数会导致生成不同的机器指令。例如，Pascal 和 Modula-2 语言中的乘法运算符“*”可同时应用于整数、实数、集合等不同类型，这些操作均采用了完全不同的机器运算来实现。语法制导代码生成不再适用于这种情况，本书将在第 8 章再适当考虑这些问题。

这里我们考虑一个更直接的问题，即变量在内存中的位置。代码生成程序必须能够输出合适的数字，以指示硬件写入或读出那些分配给特定变量的内存单元。在分析变量声明语句时，必须将变量分配到内存空间中，并且将其位置记录在符号表中以备后用。

重新审视第 5 章的 Micro-Modula 语言文法，我们发现必须扩充变量声明的非终结符 v 的属性，以包含下一个可用的变量位置的地址；并且还须扩充存储在符号表中的信息，以包含这一变量的地址。代码清单 6.1 扩展了代码清单 5.2 的属性文法，从而可以处理新的代码生成语义。

代码清单 6.1 Micro-Modula 语言的属性文法，可生成 IBSM 的代码

```
predeclared

into !SymTab !{name}int !{value}int ^SymTab;

from !SymTab !{name}int ^{value}int;

number !int {value to output};

newline;

parser                                     { 未检测标识符的类别是否合法 }

G !vacantTable:SymTab

-> "MODULE" ID ^identno " ;"
```

```

        ["3 100 0 100 -1 100"; newline]
"VAR"  V !vacantTable !3 ^tableOut ^varsOut
      O !28 O !varsOut O !4
"BEGIN" B !tableOut
      O !24 ["-1 -32"; newline]
"END"  ID ^identno "." ;

V !tableIn:SymTab !locIn:int ^tableOut:SymTab ^locOut:int

->  ID ^identno ":" T ^type ";"
    [value = type + locIn * 1000]
    [into !tableIn !identno !value ^newtable]
    V !newtable !locIn+1 ^tableOut ^locOut

->  [locOut = locIn; tableOut = tableIn] ;

T ^type:int

->  "INTEGER"
    [type = 1]

->  "BOOLEAN"
    [type = 2]
    ;

B !tableIn:SymTab

->  ID ^identno
    [from !tableIn !identno ^value]
    [loc = value / 1000]
    O !28 O !loc
    "!=" E !tableIn ^type
    O !26
    ";"
    [type = value - value / 10 * 10]
    B !tableIn

->  ;

E !tableIn:SymTab ^type:int

->  S !tableIn ^stype C !tableIn !stype ^type ;

C !tableIn:SymTab !typeIn:int ^typeOut:int

->  "=" S !tableIn ^ctype
    O !16
    [typeIn = ctype; typeOut = 2]

->  [typeOut = typeIn] ;

S !tableIn:SymTab ^type:int

->  F !tableIn ^type P !tableIn !type ;

P !tableIn:SymTab !type:int

```

```

->   "*" F !tableIn ^type
      O !11
      P !tableIn !type
      [type = 1]

->   "AND" F !tableIn ^type
      O !15
      P !tableIn !type
      [type = 2]

->   ;

F !tableIn:SymTab ^type:int

->   "(" E !tableIn ^type ")"

->   ID ^identno
      [from !tableIn !identno ^value]
      [loc = value / 1000; type = value - loc * 1000]
      O !28 O !loc O !27

->   NUM ^value
      O !28 O !value
      [type = 1]

->   "TRUE"
      O !28 O !1
      [type = 2]

->   "FALSE"
      O !28 O !0
      [type = 2]
      ;

O !value:int

-> [number !value; newline] ;

```

我们为该文法添加了一个新的非终结符 **o** 以生成输出的目标代码；它只有单个继承属性，表示待生成的值。该非终结符仅有一个空产生式，因而将它引入文法的其他地方不会对语言造成任何影响。然而，非终结符 **o** 的语义引入了两个新的语义动作例程：**number** 在输出文件中生成一个十进制整数，**newline** 生成一个行结束符。目标符号 **G** 的产生式中第一个语义动作针对 **IBSM** 初始化了输出文件，它在目标代码的开始处为栈分配了 100 个字的数据空间。具有探索精神的读者可参阅附录 C 了解 Itty Bitty 栈机器解释程序的操作细节。

与 Micro-Modula 语言扩展类型和函数声明时的做法类似(尽管此处的扩展并未包括它们)，我们将两个项目压缩为一个值，并赋给符号表中的每个符号：一个是变量的类型，另一个是它在内存中分配的位置。非终结符 **v** 分配空间的办法是让一个从左到右传递的属性加 1，该属性负责为变量的总数进行计数。在一种更复杂的语言中，我们不仅要对变量进行计数，还须计算实际分配的内存字数(或字节数，取决于机器的体系结构)；两种方法在这里是相同的。

一个新变量的位置乘以 1000、再加上表示其类型编码的整数值，即压缩为符号表中的一个

值。当一个标识符出现在一条赋值语句 **B** 或表达式 **F** 的左边时，将从符号表中查找该标识符，并解压相应的值。类型检查的方法与以前一样，而位置部分则用于生成该地址的 **LoadCon** 指令。当标识符位于赋值语句左边时，所需的只有地址；在计算赋值语句右边表达式的代码之后，输出一条 **Store** 指令。留意代码生成时的次序：首先计算将被赋值的那个变量的地址，生成的代码必须先将它压入运行时的栈中；然后为表达式生成相应的代码，并在后面接着一行 **Store** 指令将表达式的值存储到该变量中。

在因子 **F** 的产生式中，变量引用将生成一行 **LoadCon** 指令以装入地址，后面再跟着一行 **Load** 指令取出该变量的值并压入运行时的栈中。常量 **TRUE** 和 **FALSE** 只需将正确的布尔常量值（1 或 0）压入运行时的栈中即可；而整数常量则须生成代码，以压入扫描程序返回的任一数字。

注意，除了符号表中每一标识符所关联的值的压缩与解压之外，属性文法的类型检查语义并未改变。类型检查和代码生成本质上是相互独立的。

6.5.1 运算符优先级与结合性质

算术运算的标准规则规定乘法、除法运算符比加法、减法运算符有更高的优先级，即在一个没有圆括号的加法、乘法混杂的表达式中，先做乘法再做加法。这种优先级在语法文法中表现为表达式由项的和构成，项则依次由因子的积组成，而不是其他的组成方式。因而，正如我们之前为文法添加的语义，乘法是在一个项中执行的；而加法则是在一个表达式中执行，在乘法之后执行。基本上运算符在分析树中出现在越底层，则其优先级越高。

容易看出，运算符的结合性与分析树的结构有对应的关系。算术运算符通常是左结合的，因而 $a - b + c$ 解释为与 $(a - b) + c$ 相同，而不是 $a - (b + c)$ 。这导致那些缺少结合性这种代数性质的运算符处理起来不一样，譬如减法和除法运算。在文法 G_2 中，加法和乘法都是左结合的， $a + a + a$ 的分析树将左边的加法置于右边加法之下，从而为之生成的代码在执行右边加法之前，先执行了左边的加法，如图 6-4a 所示。将该文法转换为一个 LL(1) 文法时，应避免仅仅将文法翻转过来，如图 6-4d 所示，因为这会造成文法是右结合的。倘若语义动作仅将加法应用到非终结符 **T**，本书采用的构造方法保持了左结合性质，如图 6-4b 所示。对于同一文法，错误放置了语义动作也会导致加法是右结合的，如图 6-4c 所示。

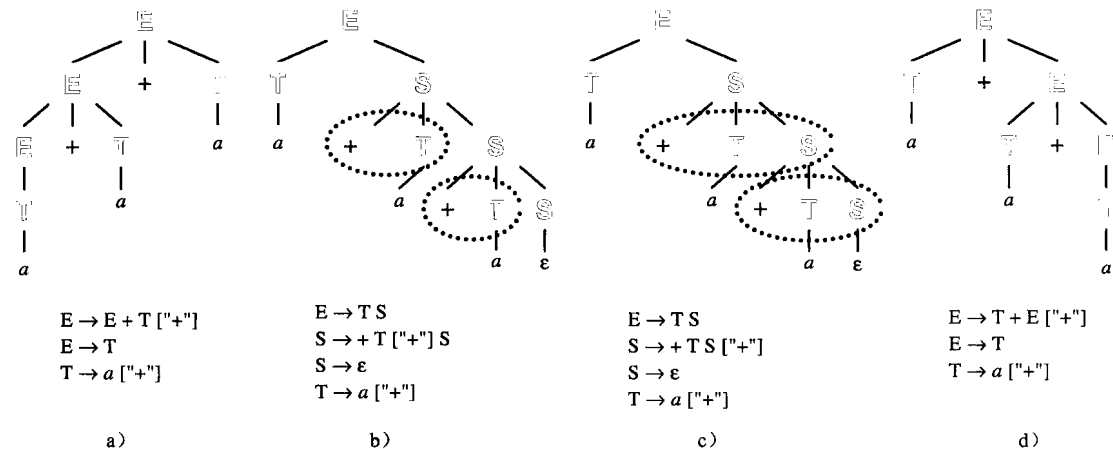


图 6-4 分析树中加法的结合性。左边的两个文法 a 和 b 是左结合的，c 和 d 是右结合的

6.5.2 程序结构的语义

如前所述，波兰表示法提供了一种简洁且正确的方法，将一个中缀表达式转换为栈机器上可执行的代码；这种方法很容易扩展到处处理简单的语句，譬如赋值和过程调用语句。程序结构，特别是 **IF-THEN-ELSE** 和 **WHILE** 循环，需要更细心的处理。

自从 Dijkstra 著名的论文“Goto 语句有害论”发表后，现代程序设计语言的设计已尽量避免非结构化跳转的用法。不幸的是硬件设计人员既不愿意也不可能听从语言设计人员的指挥，计算机硬件中的基本控制操作仍是相当于 **GOTO** 的机器实现，通常称之为分支指令。

典型的分支指令是先测试当前机器状态中的某一条件，然后若条件测试结果为真，则转去执行内存中某一指定的其他位置；否则继续执行指令序列中的下一指令。不同的机器有不同类别的条件测试，通常也会包括每一测试条件的补条件。例如，一条分支指令可能是当累加器为 0 时跳转，另一指令则会当累加器不为 0 时跳转。通常计算机有一个条件码集，条件码将不同指令（特别是单条比较指令）的结果进行编码，然后各种分支指令可测试条件码中这些位的各种有用的组合。实际上，每一种机器的设计还包括了一条无条件跳转指令，相当于隐含的条件测试结果总为真。

当分支发生时，还有不同的方法指定程序继续执行的地址。最早的硬件指定一个完整的内存地址，后来顺应指令越来越短、内存地址空间越来越大的趋势，有几种缩略形式越来越流行，这取决于它们对大多数分支的局域性的处理：程序中仅有很少的分支会跳转到离分支点很远的地方。大多数流行的分支形式是跳转到离分支指令一个固定偏移量的位置，从而在内存中重定位程序段中的指令不会导致分支地址失效。这称为相对寻址；仅含相对寻址分支的程序称为自重定位的（有时也称位置独立的），因为它们可不加修改就重定位到内存中的任何位置。

Itty Bitty 栈机器（附录 C 给出了其完整的描述）中的分支指令是相对的，因为分支出现时栈中弹出的偏移量被累加到程序计数器（即下一指令的地址）。IBSM 仅有一个条件可测试，并且该测试只有一个版本：条件是栈中的第二个字是否为 0。0 表示布尔值 **false**，分支指令仅当该值为 **false** 时才跳转，即在栈中弹出的数字为 0 时才跳转。选用三种比较运算符之一，就足以处理条件语句和所有常见形式的循环代码。IBSM 中无条件分支的组成是一条将 0 压入栈中的指令、后面接着一为 0 时分支的指令。

考虑一条简单的条件语句：

```
IF b THEN
  x
ELSE
  y
END
```

其中，**b** 是任意布尔表达式，**x** 和 **y** 是任意语句序列。在 IBSM 中实现上述语句的代码是以下指令序列和伪码，它们与所有传统计算机上的实现没有什么区别：

1	<对 b 进行求值的代码>	测试条件
2	LoadCon <偏移量 e - t>	如果为假，则跳过 then 部分
3	BranchFalse	
4	t: <执行 x 的代码>	then 部分从此处开始执行
5	LoadCon 0	跳过 else 部分
6	LoadCon <偏移量 j - e>	

```
7      BranchAlways
8  e:  <执行 y 的代码>           else 部分从此处开始执行
9  j:  <接着的任何其他代码>     两部分执行完后, 均到达这里
```

计算机必须对布尔表达式 **b** 进行求值 (第 1 行), 以判断是执行 **THEN** 部分还是执行 **ELSE** 部分。在源代码和目标代码中直接跟在后面的都是 **THEN** 部分, 因而如果条件求值为 **true** 则不会发生分支, 即分支指令在执行时不做任何事情, 只是前进到位于第 4 行的下一指令。如果表达式求值为 **false**, 就必须执行跳转到 **ELSE** 部分的分支; 这意味着分支指令会将一个偏移量累加到程序计数器, 该偏移量肯定是从分支指令后的下一指令 (第 4 行, 它也是 **THEN** 部分的开始, 其标号为 “t:”) 到 **ELSE** 部分第一条指令 (第 8 行, 其标号为 “e:”) 之间的地址之差, 这个差用形如 “e - t” 的伪码表示。**THEN** 部分 (仅当条件为真时执行) 结束后, 程序在继续执行前必须跳过 **ELSE** 部分; 这一跳转是无条件的, 因而为了跳过 **ELSE** 部分, 条件 0 作为常量被压入栈中。偏移量采用与之前相同的方法构造, 即装入一个表示汇合点 (第 9 行, 其标号为 “j:”) 与 **ELSE** 部分起点之间地址差的常量。

在这两种情况下, 偏移量都是程序中的常量; 即需跳过的指令数在编译时已固定, 因而编译程序可知道压入什么常量。然而, 编译程序若要知道这个数字的具体值, 除非先编译完所有需跳过的语句, 但这些需跳过的语句却是在使用该数字的 **LoadCon** 指令生成之后才被编译的。这就是在任何编译程序中都必须解决的向前分支问题。

6.5.3 向前分支问题

向前分支问题有两种解决途径, 这两种途径都较常见的。其中一种方案是两次编译该程序, 仅在第二次遍历时才生成代码; 第一次遍历时找出并记录所有的分支目标, 以备第二次遍历时使用。采用该方案的编译程序称为 “两遍” 编译程序, 因为它两次遍历了该程序。该方案的一种变形是将所有不完整的向前引用转换为链接程序中的特殊命令, 并依靠后续的 “链接—重定位程序” 步骤解析这些向前引用。尽管两次读入源程序文本会消耗更多的时间, “两遍” 编译程序在编译不要求 “先声明、后使用” 的语言时是非常必要的, 因为在第一次遍历中并不能总是知道任一标识符的引用会生成什么代码。

第二种方案是在编译时保留每一个不完整的向前分支记录, 然后在目标地址变为已知的时候, 在输出代码中回到原来位置并为偏移量或分支地址填写正确的值。该方法称为回填, 基于该方法有可能构造出一个 “一遍” 编译程序。我们采用 **Pascal** 和 **Modula-2** 这类块结构语言帮助理解这一方法, 尽管这类语言要求在输出文件中有一些非顺序性。

所有地址偏移量的计算都必须基于已生成的指令的地址, 而这些指令反过来又要求代码生成产生式自始至终将该地址偏移量的值作为随身携带的属性。一种可行的办法是综合出一个表示大小的属性, 当该属性沿分析树向上流动时累加其值, 并且该属性可为生成一个分支的相对偏移量提供足够的信息。然而, 我们更喜欢从左到右的绝对地址属性, 从而在我们以后为符号表提供一个过程的绝对地址时也可用到该属性。为 **IF** 和其他语句生成代码的属性文法片段 (不含类型检查) 如代码清单 6.2 所示; 在该文法中, 非终结符 **Emit** 负责维护输出目标代码时的位置计数器。

代码清单 6.2 为 IF 语句生成回填代码

```

Stmt !locIn:int ^locOut:int

-> "IF" BoolExpn !locIn ^locEx
    Emit !28 !locEx ^locOff1
    Emit !0 !locOff1 ^locBrf
    Emit !1 !locBrf ^locThen
    "THEN" Stmts !locThen ^locLdz
    Emit !30 !locLdz ^locLdc
    Emit !28 !locLdc ^locOff2
    Emit !0 !locOff2 ^locBra
    Emit !1 !locBra ^locElse
    "ELSE" Stmts !locElse ^locOut "END"
    BackPatch !locOut !locOff1 !locElse-locThen
    BackPatch !locOut !locOff2 !locOut-locElse

-> OtherStmt !locIn ^locOut ;

BackPatch !locn:int !locOff:int !value:int

-> ["-1 "; number !locOff; newline]
    [number !value; newline]
    ["-1 "; number !locn; newline] ;

Emit !value:int !locIn:int ^locOut:int

-> [number !value; newline; locOut = locIn + 1] ;

```

假设该编译程序为一个标准 IBSM 装载程序生成代码；该装载程序在识别到一个“-1”后面跟着一个数字时，会将装载地址重定向为该数字。因此，我们不必回退或回卷输出文件，也不必将输出文件整个缓存在内存中。非终结符 **BackPatch** 在指定的地址生成一个字（使用该地址重定向），然后将地址位置计数器恢复为当前地址（通过第一个继承属性 **locn** 传递给它）。

试看该文法如何编译一个小程序片段。给定以下语句序列：

```

a := 3;
IF a < b THEN
    b := 1
ELSE
    b := 5
END;
c := 0;

```

假设当前位置计数器为 147，且 **a**、**b** 和 **c** 的局部地址分别为 11、12 和 13。考虑分析程序正准备接受单词 **IF** 时的场景，此时输出文件如下所示（为更清晰地表示，同时显示了位置计数器的值和指令助记符，尽管它们不会出现在输出文件中）：

...			
(147)	28	LDC 11	(装入 a 的地址)
(148)	11		
(149)	28	LDC 3	(装入常量 3)
(150)	3		


```
(151)      26          ST          ( 将常量 3 存储到 a )
(152)
```

现在分析程序处理到非终结符 **Stmt**，且 **locIn** 为 152。下一单词是 **IF**，故扫描程序接受该单词后，分析程序以继承属性 **locIn** = 152 调用非终结符 **BoolExpn** 以分析布尔表达式 **a < b**，从而生成如下代码：

```
(152)      28          LDC 11      ( 装入 a 的地址 )
(153)      11
(154)      27          LD          ( 装入 a 的值 )
(155)      28          LDC 12      ( 装入 b 的地址 )
(156)      12
(157)      27          LD          ( 装入 b 的值 )
(158)      17          LESS        ( 执行比较 )
(159)
```

现在从 **BoolExpn** 回到非终结符 **Stmt**，所携带的综合属性 **locEx** 值为 159，此时又多生成三条指令字：

```
(159)      28          LDC 0        ( 偏移量的占位符 )
(160)      0
(161)      1          BRF          ( 为假时分支 )
(162)
```

综合属性 **locOff1** 的值为 160，地址 0 最终将被向前分支的偏移量取代。综合属性 **locThen** 的值为 162，即 **IF** 语句中 **THEN** 部分的起始地址，同时也是计算分支偏移量时必不可少的基地址。如果我们在 **IBSM** 上运行这些代码并观察栈中的内容，它将具有如图 6-5 所示的值（设 **b** 为 4）。

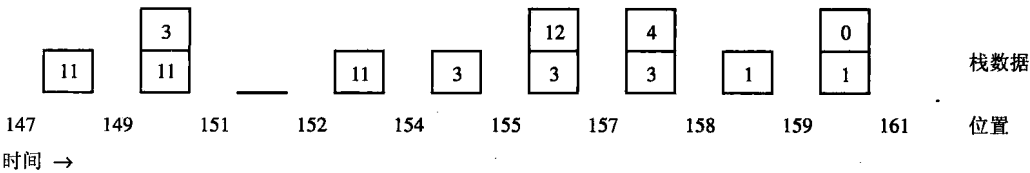


图 6-5 语句序列 “a := 3; IF a < b THEN ...” 的执行轨迹

接着以继承属性 **locThen** 中的当前地址 162 递归地调用非终结符 **Stmts**，为赋值语句生成代码，生成代码后综合属性 **locLdz** 的求值结果为 167。另外再生成 4 个字的代码后，输出文件形如：

```
(162)      28          LDC 12      ( 装入 b 的地址 )
(163)      12
(164)      28          LDC 1        ( 装入常量 1 )
(165)      1
(166)      26          ST          ( 将常量 1 存储到 b )
(167)      30          ZERO        ( 将 FALSE 压入栈中 )
(168)      28          LDC 0        ( 另一偏移量的占位符 )
(169)      0
(170)      1          BRF          ( 实现 BranchAlways，即绝对跳转 )
(171)
```

留意 IBSM 并没有真正的 **BranchAlways** 指令，这一效果是通过 0 和 **BranchFalse** 指令合成的。执行到这里，分析程序将接受单词 **ELSE** 并返回非终结符 **Stmts**，同时携带了一个新的继承属性 **locElse**，其值为 171。尽管此处并未给出完整的规格说明，但是非终结符 **Stmts** 会将该属性看作自己的一个继承属性(可能命名为 **locIn**)，正如它之前以同样方式看待值 162；在生成 **b := 5** 的代码后，返回一个综合属性（在 **Stmt** 中以 **locOut** 命名），其值为 176：

```
(171)      28          LDC 12          ( 装入 b 的地址 )
(172)      12
(173)      28          LDC 5           ( 装入常量 5 )
(174)      5
(175)      26          ST              ( 将常量 5 存储到 b )
(176)
```

到了这里，正是回填技术最吸引人的地方。非终结符 **BackPatch** 被指定了三个继承属性：第一个是当前位置，从而它可在回填后恢复指令序列顺序中的下一个字的装入地址；第二个属性是待回填的内存地址；第三个属性是将填入该位置的值。局部属性 **locOff1** 和 **locOff2** 分别保留了两个占位符 0 的地址，每一个占位符均需调用一次 **BackPatch**。据前述在 IBSM 上实现 **IF** 语句的伪码，第一次回填需要 **ELSE** 部分与 **THEN** 部分的地址之差，即 $171 - 162 = 9$ 。在最后的代码生成后，**BackPatch** 为这三行在输出文件产生以下代码：

```
(175)      26          ST              ( 将常量 5 存储到 b )
           -1 160      ( 偏移量的地址 )
(160)      9           ( 偏移量的值 )
           -1 176      ( 恢复当前地址 )
(176)
```

第二次调用非终结符 **BackPatch** 时，继承属性 **locOff** 的值为 169（地址存放在 **locOff2**），且 **value** 的值为 5（即 $176 - 171$ ）；故又有以下三行添加到输出文件中：

```
           -1 169      ( 偏移量的地址 )
(169)      5           ( 偏移量的值 )
           -1 176      ( 恢复当前地址 )
```

最终，非终结符 **stmt** 退出，其综合属性 **locOut** 的值为 176，即待生成的下一机器指令的地址。接在 **IF** 语句后的赋值语句又生成 5 个字的代码，结果前述三条高级语言语句的输出文件片段形如（以 3 列展示，不再显示注释）：

```
28          1          26
11          28         -1 160
28          12         9
3           28        -1 176
26          1         -1 169
28          26         5
11          30        -1 176
27          28         28
28          0          13
12          1          28
27          28         0
17          12         26
28          28
0           5
```

然而，如果在上述文件装入内存后按传统方法打印内存，可看到偏移量的值被正确地设置，类似如下（内存地址和数据按十进制显示）：

0147	28	11	28
0150	03	26	28	11	27	28	12	27	17	28
0160	09	01	28	12	28	01	26	30	28	05
0170	01	28	12	28	05	26	28	13	28	00
0180	26						

基本的 **WHILE** 循环语句只是引入了“向后分支”这一新概念。属性求值函数必须计算分支的偏移量（为负数），但由于这两个位置在代码生成时均已固定，因而并不需要回填技术。当然，循环条件求值为假时，向前分支仍要求回填以保证其偏移量被正确地设置。对于以下循环：

```
WHILE b DO
    x
END
```

生成的代码应如下所示：

t:	<对 b 进行求值的代码>	循环前的测试条件
	LoadCon <偏移量 e - b>	如果为假则退出
	BranchFalse	
b:	<执行 x 的代码>	执行循环体
	LoadCon 0	跳回循环的开头
	LoadCon <偏移量 t - e>	
	BranchAlways	
e:	<接着的任何其他代码>	循环结束后到达这里

实现上述代码的属性文法产生式留给作为练习。

6.6 过程和函数的代码生成

大多数传统的过程式程序设计语言中，过程与函数的区别仅在于是否有返回值；除此之外，参数（如果有的话）的求值方式是相同的，调用序列是相同的，进入和退出过程或函数的代码也是相同的。可能正因为如此，沃思将 Pascal 语言改进为 Modula-2 语言时，不再将 **FUNCTION** 列为单独的关键词。

早期计算机体系结构的一大创新（回到真空管时代）是子例程（过程）调用的概念，其中硬件负责保存子例程结束后返回的指令地址。事实上，每一种现代计算机的指令集现在都包含两条指令，以实现高效的子例程调用与返回。**Call** 指令（有时也称子例程跳转指令）将下一指令的地址保存在某一指定的位置（通常是在硬件的栈中，但很多时候改为在寄存器中），然后无条件跳转或分支到一个子例程的地址；**Return** 指令将程序控制恢复为所保存的地址。现代高级语言通常还需要另外的辅助指令负责传递参数、在子例程中为局部变量分配存储空间、允许递归等。有三个代码序列值得特别关注：子例程的进入、退出和调用。一个给定的子例程往往不只一次被调用，因而将进入和退出代码尽量集中在子例程之中是更有利的，而不是在每一调用序列中复制这些指令。

进入一个过程大体上与进入一个程序是相同的，但没有初始化输出代码模块的代码文件设置语义，取而代之的代码是建立一个称为 **Display** 表的指针数组。每一指针包含了当前过程的外围过程中的局部变量空间的地址。在 C 和 FORTRAN 这类非嵌套型语言中，**Display** 表仅需

两个入口：局部变量和全局变量，因而节省了初始化设置的大部分开销。在 IBSM 中，构造一个 Display 表涉及将 33 条指令压缩为 16 个字。然而，代码生成的语义并不关心构造 Display 表的技术细节（有一份类似菜谱的说明就足以满足我们的需要）。子例程的退出代码仅有两条指令，第一条指令将一个表示形式参数个数的常量压入栈中，第二条指令将实现子例程的退出。与硬件细节有关的进入和退出代码跟这里的关系不大，况且每一计算机都是不同的。有兴趣的读者可参阅附录 C 中关于 Itty Bitty 栈机器过程进入和退出的完整代码。

一个过程的调用序列由以下指令组成：对每一值参（如果有的话），将其表达式值压入栈中；接着将被调用过程的地址及其父过程的帧指针压入栈中；然后是一条 Call 指令。父过程的帧指针用于构造 Display 表，以支持对非局部变量的访问。

函数调用会在参数压栈之前，先将返回值的空间压入栈中。函数还有一条 RETURN 语句，该语句必须将待返回的值保存到由调用者在栈中保留的函数返回结果空间中，然后退出函数。图 6-6 展示了 IBSM 中一个函数过程的调用、进入和退出代码的常见结构。

主程序代码	函数过程代码
·	abc: LoadCon n (分配局部变量)
·	Enter (并创建局部帧)
·	构造 Display 表
LoadCon 0 (返回结果的空间)	过程代码的起点
将参数压入栈中	·
将父帧指针压入栈中	·
LoadCon abc (子例程地址)	保存函数的返回结果
Call (执行函数)	·
使用返回结果	·
·	过程代码的结束
·	LoadCon k (待弹出的参数个数)
·	Exit (返回调用者)
·	

图 6-6 IBSM 中的函数调用、进入和退出代码

6.7 块结构的栈帧管理

在 Modula-2 或 Pascal 这类块结构语言中，一条语句不仅可引用直接包含了该语句的过程中的局部变量，还可引用包围了其直接过程的任一外围过程中的所有变量。第 5 章学习了这种结构如何影响到符号表的管理，现在须考虑正在执行的程序如何能访问到这些变量（从而可装入或存储这些变量中的值）。像 C 和 FORTRAN 这类非块结构语言没有这一问题，因为在任何时候都仅有两个层次是可见的：全局的（在 FORTRAN 语言中称为 COMMON）或局部的；所有过程均不可访问任何其他过程中的变量，除非将引用作为参数传递。

6.7.1 帧与帧指针

每一过程或函数在进入时（即它被调用时），会分配一些内存以存储它的局部变量。这一内存块称为“帧”，它通常具有固定的大小，因为在编译时可确定有多少变量使用了多少个字的内存。被递归调用的过程在每次实例化时都会分配一个新的帧，因而每一调用实例的变量之间不会相互影响。

有些语言允许使用静态变量，它们并不在局部帧中分配空间；这些变量实际上是全局变量，但仅在它们被声明的过程中才是可见的。由于除可见性之外，它们的作用与全局变量相同，此处对它们不作特别考虑也是无妨的。FORTRAN 语言的大多数实现是让所有变量都是静态的。

局部变量帧通常由一个硬件实现的帧指针来访问，该指针要么是一个专用于此目的的特殊寄存器（正如 Itty Bitty 栈机器的设计），要么是一个习惯上用于此目的的通用地址寄存器。于是局部变量由从当前帧指针开始计算的固定偏移量寻址，我们保存在符号表中的正是这些偏移量（与变量的类型和类别同时保存）。局部帧通常还包含由过程调用者保存的寄存器（亦称“状态”）。这些帧通常动态地分配，并链接在一起形成一个链表；要么在一个栈中顺序地分配。

图 6-7 展示了一个嵌套了过程 A 和 B 的主程序所用的局部变量帧的链表。当一个过程或函数终止并返回其调用者时，该过程或函数将释放其局部帧，并将硬件指针恢复为指向其调用者的帧，同时恢复原来保存的所有其他状态（例如寄存器）。在 IBSM 中，这一过程由单条 **Exit** 指令基本上自动完成。

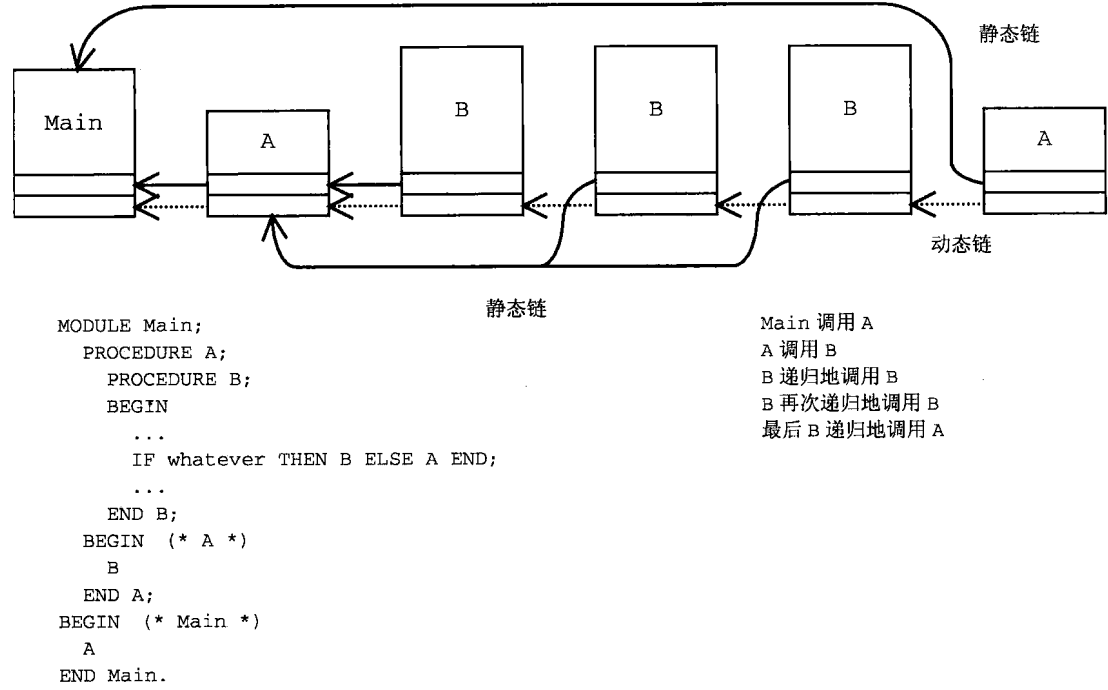


图 6-7 变量帧的一个链表；同时展示了程序代码清单，以及拍摄帧快照这一时刻的程序执行轨迹

6.7.2 静态链与动态链

在 C 语言这类递归的、但非块结构的语言中，只有全局变量（往往分配在内存中的固定位置，或通过另一专用寄存器访问）和局部帧。而像 Modula-2 这类块结构语言中，通常不仅需要访问局部变量和全局变量，而且还必须能够访问中间层过程的局部变量。程序必须获得一个过程的帧指针，方可访问该过程中的这些变量。链表中的下一帧表示了调用者的顺序，但它们很多时候并不能帮助访问包围了该过程作用域的上一层作用域。

例如，如果一个过程递归地调用自身，就会有同一过程的几个帧链接在一起，如图 6-7 中过程 B 的帧所示；然而对上层（非局部）变量的引用在查找包含被引用变量的帧时（可能在过程 A 中），必须绕过所有这些帧。因而，帧通常还用另一个链表连接在一起，用于表现作用域可见性的链接。这一条链称为静态链，而调用者的链则称为动态链。

像 Lisp 这类动态作用域语言仅使用动态链访问非局部变量，但由于这类语言的编程充满了

潜在错误，所以很少语言会背离 Pascal 和 Modula-2 这类语言的静态作用域规则。实际上，Lisp 语言的一些变形已回归到静态作用域。

6.7.3 帧指针的 Display 向量

访问非局部变量的最简单办法，是在每一过程入口处构建一个帧指针的向量，称之为 Display 表。在一个静态确定作用域的语言中，词法层次是指源代码文本中所包含的过程的数量，这在编译时即可确定。因而 Display 表是一个固定长度的数组，其元素个数与当前的词法层次相同，每一元素按词法层次顺序指向一个所包含的过程的帧指针。

在 IBSM 中，Display 表并不是由指针组成，而是由从当前帧指针开始计算的（用一个负数表示）偏移量组成。构建 Display 表的 IBSM 指令顺列并没有很大的指导意义，因而此处仅以十进制绝对值展示这些代码，有兴趣的读者可参阅附录 C 中的完整代码清单。代码清单 6.3 是 Micro-Modula 语言属性文法的一个片段，展示了如何设置无参数函数的头部（为表述清晰，我们再次省略了类型检查）；过程头部与此仅有少许差别，其不同之处是显而易见的。

代码清单 6.3 无参数函数的过程头部属性

```

H !tblIn:SymTab !locIn:int !lex:int ^tblOut:SymTab ^locOut:int

->  "PROCEDURE" ID ^identno ":" T ^type ";"
    [value = type + locIn * 1000 + lex * 100 + type + 40]
    [into !tblIn !identno !value ^nexttable]
    [open !nexttable ^newtable]
    "VAR" V !newtable !0 !lex+1 ^vtable ^nvars
    Emit !30 !locIn ^loc1 \      { 跳过嵌套的函数 }
    Emit !28 !loc1 ^loc2
    Emit !0 !loc2 ^loc3
    Emit !1 !loc3 ^locn
    H !vtable !locn !lex+1 ^btable ^locd
    BackPatch !locd !loc2 !locd-locn
    Display !locd !nvars !lex+1 ^locb
    "BEGIN" B !locb !type !lex+1 !nvars ^locx
    Emit !28 !locx ^loc6      { 即 LoadCon nargs + 1 指令 }
    Emit !1 !loc6 ^loc7
    Emit !5 !loc7 ^locz
    "END" ID ^identno ";"
    H !nexttable !locz !lex ^tblOut ^locOut

->  [locOut = locIn; tblOut = tblIn] ;

Display !locIn:int !nvars:int !lex:int ^locOut:int
                                     { 建立过程或函数入口，并分配局部变量 }

->  Emit !28828 !locIn ^loc1      { 总共 15 字 }
    Emit !nvars !loc1 ^loc2
    Emit !lex !loc2 ^loc3
    Emit !44968 !loc3 ...
    [继续输出以下数字: 13288, 30, 21481, 268, 1149, 26473, 7102, 52, 32044, 31124]
    Emit !53661 !loc14 ^locOut ;

```

在一个过程或函数之中，先用 **LoadCon** 读入地址偏移量后，再接着一条 **Load** 或 **Store** 指令，仍可像以前那样访问局部变量。在 IBSM 中，所有内存均通过当前过程的帧的相对位置

来引用，该帧是在过程的入口处建立的；非局部变量仅可间接地通过 Display 表访问。由于在语法上无法区分局部变量与非局部变量，我们同时也为局部变量生成 Display 表的访问代码。构造一个局部变量或非局部变量地址的 IBSM 指令序列是：

LoadCon <offset> { 变量在它所属过程的帧中的位置 }

LoadCon <display + lexlevel> { 指向 Display 表的下标 }

Load { 取变量所在帧的偏移量 }

Add

```
MODULE Demo;
VAR
  a, b, c: INTEGER;

  PROCEDURE x(): INTEGER;
  VAR
    s, t: BOOLEAN;

    PROCEDURE y(): BOOLEAN;
    VAR
      v: INTEGER;
    BEGIN (* y *)
      v := x();
      RETURN b = v
    END y;

    BEGIN (* x *)
      s := a < b;
      a := a + 1;
      IF s THEN t := y() END;
      RETURN a - b
    END x;

  BEGIN (* Demo *)
    a := 1;
    b := 2;
    c := x()
  END Demo.
```

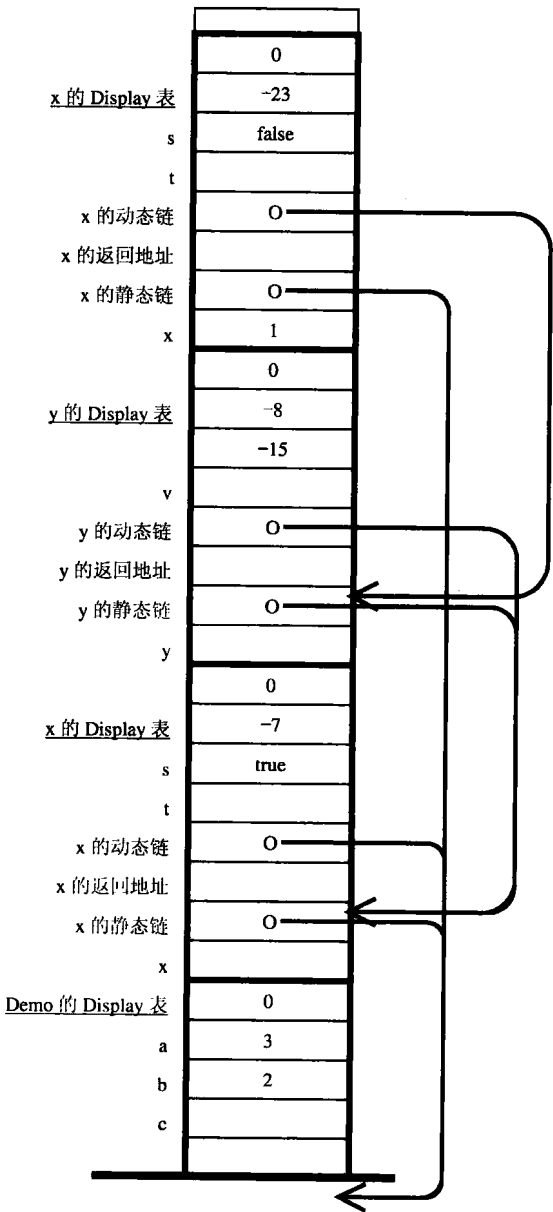


图 6-8 递归调用过程中的 IBSM 栈。该快照展示了对过程 x 的内层调用中，执行到那条用方框标注的 RETURN 语句时的栈

符号表为该变量指明了当前的 offset 和 lexlevel; lexlevel 为 0 表示全局变量，为

1 表示所包含过程的第一层中的变量，如此类推。通常 **lexlevel** 作为一个继承属性在分析过程中传递，并且当一个变量被声明时被存入符号表中。常量 **display** 表示在局部帧中 **Display** 表的相对地址；它等于局部变量的数目加 3，因为在局部帧中，局部变量的空间分配在 **Display** 表之前。当这一指令序列后面跟着一条 **Load** 指令时，变量值将被取出并用于表达式中；相反，如果后面跟着一条 **Store** 指令，则栈中恰好在偏移量之上的值将被存入寻址得到的变量。

为被调用过程在静态链上的“下一链接”设置一个正确的指针，这是由调用者负责的功能，我们称此指针为“父指针”。父指针指向当前正被调用的过程的下一静态外层的帧指针。如果是自身的递归调用，则下一外层与调用者的下一外层相同；如果是调用与自身位于同一词法层次的兄弟过程，则父指针与调用者的父指针相同；如果当前正被调用的过程被包含（嵌套）在调用者中，则父指针就是调用者自身的帧指针；否则，是一个递归的调用，或是调用沿作用域链向上一层或多层的“叔伯”过程。图 6-8 展示了几次过程调用（包括一次递归调用）后的栈。

在每一种情况下，父指针都根据当前的 **Display** 表计算，并在调用过程或函数之前压入栈中。调用一个过程的指令序列是：

Zero	{ 为返回值创建空间 }
<如果有参数则压入所有参数>	
Zero	{ 计算自身的帧指针的值, }
Global	{ 即 $-(0 - FP)$ }
Negate	
LoadCon <display + lexlevel>	{ 指向 Display 表的下标 }
Load	{ 取父指针的偏移量 }
Add	{ 将父指针留在栈中 }
LoadCon <过程的地址>	
Call	{ 跳转到该过程 }

符号表中过程名字的词法层次可用于检索 **Display** 表以取得该过程的父指针的正确偏移量。前面的三条指令将当前帧指针转换为栈中的一个字，这些代码用于处理一个具有返回值的函数；在调用一个无返回值的过程时，不应压入第一个 0。

图 6-8 仅以图形方式展示，实际上过程 **x** 的静态链有一个不那么直观的值 -1，这是其父过程（即主程序）的帧指针。全局变量是从这一帧开始的偏移量，该帧由主程序中的初始 **ENTER** 指令设置为比当前的（初始的）栈指针值（即 0）小 1。还应留意到，在单条声明语句中用逗号分隔的多个变量将以逆序分配到栈中，这是递归分配方案（参阅第 5 章练习 1（d）和 1（e））的自然结果。

6.8 其他数据类型

迄今为止，我们仅考虑了变量类型 **INTEGER** 和 **BOOLEAN**，每一类型占用一个字的内存。通常还会用到许多其他类型，包括复杂数据类型。标准 **Modula-2** 语言表示了其中的大多数类型，因而我们基于这一背景讨论此课题。

除 **INTEGER** 和 **BOOLEAN** 类型外，**Modula-2**（以及 **Pascal**）语言提供了三种另外的简单数据类型：**CHAR**、枚举类型和 **REAL**。**BOOLEAN** 类型是枚举类型的一种特例，因为编译程序不必显式地引用类型名字就可从关系表达式产生 **BOOLEAN** 类型的值，条件表达式（**IF**、**REPEAT**

和 **WHILE**) 也需要 **BOOLEAN** 类型的值; 除此之外, **BOOLEAN** 类型遵循任何枚举类型的一般规则。特别是所有枚举类型均由常量名的有序列表组成, 它们实际上是常量标识符, 但在我们限制的语言中强制规定了 **BOOLEAN** 类型中的常量名是保留字。

常量名通常作为常量登记到符号表中, 链接到符号表中该类型的入口。枚举类型的常量标识符被赋值为从 0 开始的顺序值 (正如 **FALSE** 为 0, 而 **TRUE** 为 1), 代码生成的语义将它们作为数值常量处理。不幸的是, 语法上的二义性导致无法区分常量和变量标识符 (即同一语法结构 “标识符” 会产生不同的代码), 这限制了语法制导代码生成对常量标识符的处理。由于这一原因, Itty Bitty Modula 语言放弃了通用的枚举类型。

标量类型 **CHAR** 不存在语法二义性问题, 其实现较为直接。**CHAR** 类型的常量在语法上通过括上引号加以区别。除了常量的值等于 **ASCII** 字符编码 (或适合硬件和操作系统的任何其他编码) 之外, 为实现字符常量而生成的代码与实现数值常量的代码相同。除非通过存储在符号表中的类型码, 否则 **CHAR** 类型的变量无法与其他标量类型的变量区别开来。强类型检查要求类型码必须是一致的, 但生成的代码不必区别它们; 当然, 如果分析一个 **CHAR** 类型的操作数, 对一个算术运算符或布尔运算符的类型检查会发现一个错误。

标量类型的子界问题稍微复杂一些。子界类型的变量通常分配了与其基类型一样完整的内存空间, 但它们必须作为一个子界类型登记到符号表中, 从而可为它们生成范围检查代码。然而, 类型检查只能参考它们的基类型。当编译程序被限制为严格的语法制导语义时 (正如本书迄今为止的做法), 有必要将每一标量类型作为子界类型登记到符号表中; 基类型只是其子界为最大范围的类型。编译程序会为每一条赋值语句生成范围检查代码, 不管它是否有需要。第 9 章将介绍可消除不必要的范围检查代码的优化方法。

标量变量通常会分配一个字的内存。在以字节寻址的机器中, 可能更方便的做法是为枚举和字符变量以及范围较小的 **INTEGER** 分配单个字节。如果硬件对更大单元的奇地址有限制, 则编译程序在分配内存空间时必须小心处理必要的对齐, 即插入无用的字节作为填充符。由于每一变量均属于一个指定的类型, 符号表中的类型入口除其他东西外还应包含该类型的一个变量在内存中需占用多少字或多少字节的信息, 可能还有一个标志位表示是否有必要实现字的对齐。通常认为字对齐对填充到一个或多个字中的任意类型都是必要的, 因而在这种情况下标志位可以省略, 取而代之的是依靠大小来判断是否有字对齐的需求。

浮点变量 (大多数语言中的 **REAL** 类型) 所占空间通常多于用于存储整数的单个字; 如果情况如此, 它们会自始至终地占用这么多的空间。除重载了算术运算符的情况之外, **REAL** 变量可看作一种特殊的不透明记录, 即它们占用多个字的内存并作为一个整体移动; 它们有自己的语义 (为浮点运算生成的代码很少与整数的相同), 因而我们又遇到二义的语义问题。由于浮点类型表达式求值的代码生成与整数的类似, 故此处不再赘述。

6.9 结构化数据类型

借助于三种主要的数据结构, 程序员可通过适当的类型定义构建内存中变化无穷的数据组织方式。在现代程序设计语言中, 可显式地控制对每一结构中元素的访问, 因而编译访问这些结构的所有语义都可以是语法制导的。这三种结构分别是: 数组、记录和指针。它们可以混合和嵌套至任意深度, 但一个属性文法的改进 **PDA** 通过匹配符号表中的声明, 可轻易走出访问

运算符的迷宫。

为分析所谓的 L 表达式，通常最实用的办法是在文法中为它定义单个非终结符。一条 L 表达式表达的语义是访问一条赋值语句左边（L 表达式中的“L”表示 Left）的变量，可看作一个返回变量地址的内联函数。在 Itty Bitty 栈机器中，L 表达式表示了访问一个变量时所需的所有代码，最后的 **Load** 或 **Save** 指令除外。仅考虑三种主要结构以及原始变量类型（仅占 1 个字），L 表达式的语法如下：

$$\text{Lexpn} \rightarrow \text{ID} (\text{"^"} | \text{"."} \text{ ID} | \text{"[" Expn "]" })^*$$

由此可见，一条 L 表达式由一个标识符接着任意数目、任意次序的结构访问运算符组成。这三种结构访问运算符分别是指针析取、记录字段引用以及数组下标。由于它们是独立的，且每一运算符均生成语法制导的代码，故可分别处理这三个运算符。

6.9.1 指针类型

最简单的结构化数据类型是指针类型。直观地看，指针是一个巨型数组（即全部内存，或至少是用于分配动态变量的整个堆）的下标。尽管内存中的指针相当于一个大整数（在 IBSM 中占 1 个字），指针类型有别于它所指向的类型。然而，在符号表中必须有从指针类型到其基类型的链接；图 6-9 展示了一种可能的实现。

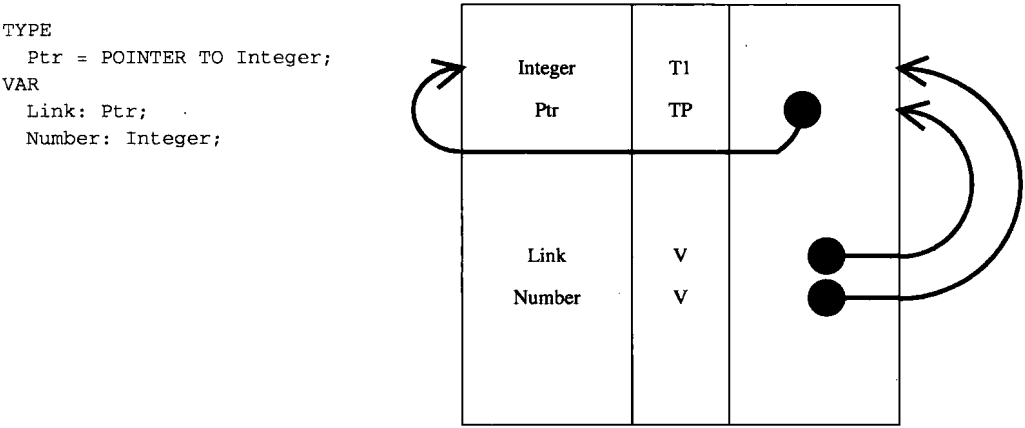


图 6-9 符号表中的指针类型

L 表达式中，指针的语义相对较为容易。每次析取一个指针（即每次出现析取运算符“^”）时，当前类型也必须被析取。在图 6-9 中，对标识符 **Link** 的引用属于 **Ptr** 类型，但 **Link^** 的引用却属于 **Integer** 类型。对于指向指针的嵌套指针，可根据嵌套链的长度作相应次数的析取，也可根据声明插入另外的结构运算符；每次析取时，均须检查正在析取的类型必须是指针，然后将新的待处理 L 表达式的类型设置为析取后的类型。

析取一个指针的代码也较为简单，每一析取运算符增加了一次从全局内存中取数据。在 IBSM 中，内存引用相对于帧指针是局部的，而 **Global** 指令可将它转换为全局的。因而，析取运算符在 L 表达式中恰好生成两条指令：

(LoadCon a)
...
1 Load
2 Global

{ 取指针 a 的地址 }

{ 析取指针 }
{ 将局部地址转换为全局地址 }

```

...
(Load)                                { 取出变量 }

```

6.9.2 记录结构

记录结构类似一个缩小的帧。声明一个记录时，必须在符号表中创建一个新的作用域，并将记录中的字段登记为这一私有作用域中的标识符。每一字段均被标注为字段类别，而所有其他的语义则与局部变量的声明相同。更具体地说，一个局部变量的偏移量指的是该变量在局部变量帧中的位置，而一个字段的偏移量则是指该字段相对于记录的位置。因而，记录中第一个字段的偏移量为 0，第二个字段的偏移量等于第一个字段的大小，如此类推。一个记录类型的对象的大小是所有字段大小的总和，这一个合计大小应与记录的定义一起保存在符号表中。

在记录声明的最后将关闭作用域，并且将整个记录声明保存到某一位置，要么是符号表中某一不可访问的部分，要么是符号表之外的某个地方。当一条 L 表达式通过句号运算符访问一个记录的字段时，已保存的作用域被打开，并且字段名的符号表查找被定向到那个重新打开的作用域，而不是整个可见的符号表。当然，到目前为止被分析的 L 表达式的类型必须是记录，并且符号表中的记录类型必须链接到已保存的字段列表。在所有其他的方面，记录字段的类型检查语义处理与变量标识符的处理相同。分析一个字段名时，其类型（从符号表中的字段声明可知）成为当前 L 表达式的待处理类型。

在 IBSM 中，为一个记录字段的引用生成的代码又恰好是两条指令。第一条指令装入一个常量，即该字段在记录中的偏移量；第二条指令将它累加到正在处理的 L 表达式的地址：

```

(LoadCon a)                            { 取记录的地址 a }
...
1  LoadCon <offset>                    { 取字段的偏移量 }
2  Add                                  { 将它累加到记录的地址 }
...
(Load)                                { 取出变量 }

```

在基于寄存器寻址的传统计算机上，只要将偏移量累加到用于访问变量的寄存器即可访问记录的字段。这将运行时的加法转换为编译时的加法，本质上是一种代码优化。第 8 章讨论代码优化时将看到，类似的优化也可用于 IBSM 代码。

6.9.3 数组的语义

数组比指针和记录稍复杂一些。一个数组是内存中的分量的线性序列，通过下标类型进行编号。因而，一个数组的定义必须指定两个子类型：下标类型和分量类型。下标类型必须是一个标量类型（整数、布尔值、字符、任意枚举类型，或它们之中的子界类型），而分量类型则可以是任意类型。一个数组定义的符号表入口必须同时链接到这两个类型。下标类型必须指定数组的下界以及元素的个数（或以等价的方式指定其上、下界）；所有标量类型都必须包括这些上界和下界，从而可实现某种方式的范围检查。分量类型必须指定该类型中一个对象的大小；其实以任一类型声明的变量都毫无例外地有这一要求。在所定义的数组类型中，一个对象的大小是下标类型中的元素个数与分量类型大小的乘积。

数组引用的类型检查较为简单，只要保证正在处理的 L 表达式是一个数组类型，并检查下标表达式属于与数组声明的下标类型相同的类型即可。数组的分量类型成为正在处理的 L 表达式的新类型。

为数组元素的引用而生成的代码依赖于符号表中存储在数组定义中的全部语义信息。正在处理的 L 表达式求值已经计算出该数组的地址，这一地址与数组第一个元素的地址相同。计算一个简单的下标操作 $a[k]$ 需要十几条不同的机器运算。如果我们先将下标表达式转换为等价的伪码，将更容易理解这些问题。给定如下定义的数组：

```
VAR
    a: ARRAY [lo .. hi] OF whatever;
...
    a[k]
...
```

我们必须先对下标表达式 k 求值，以找出数组的第 k 个元素。由于数组下标以常量 lo 为基数（它可能不是 0），故必须从 k 计算得到的值中减去这一基数，从而将第 k 个元素的偏移量规范化。然后，一个类型安全的编译程序还将校验下标的范围：它既不可小于 lo ，也不可大于 hi 。如果存储在符号表中的下标范围表现为一个基数和元素的个数，则在减去基数后再执行这一测试会更高效。最后，用分量类型大小（以可寻址的字或字节数计算，这取决于特定的硬件）乘以规范化的下标，从而确定该元素的物理偏移量。这一计算过程看起来类似如下的类 Modula-2 伪码：

```
PROCEDURE SubscriptCalc(A: ADDRESS; k, lo, nelts, eltsize: INTEGER): ADDRESS;
VAR
    temp: INTEGER;
BEGIN
    temp := k - lo;
    IF temp < 0 THEN
        Error
    ELSIF temp >= nelts THEN
        Error
    ELSE
        RETURN A + (temp * eltsize)
    END
END SubscriptCalc;
```

相同操作的抽象 IBSM 代码形如：

(LoadCon a)	{ 取数组的地址 a }
...	
1 <Expn>	{ 对下标表达式进行求值 }
2 LoadCon <lobound>	{ 减去下界 }
3 Negate	
4 Add	
5 Dupe	{ 为范围检查准备 2 个副本 }
6 Dupe	
7 Zero	{ 保证它既不会小于 0, }
8 Less	
9 Swap	
10 LoadCon <numelements>	{ 也不会大于元素个数 }
11 Greater	
12 Or	{ 如果其中一个为真, }
13 LoadCon 1	
14 BranchFalse	

```

15 Stop                                { 遇到错误则停机 }
16 LoadCon <componentsize>          { 计算偏移量 }
17 Multiply
18 Add                                { 累加到数组的地址 }
...
(Load)                                { 取出分量 }

```

为一个指定元素寻址的代码必须添加到下标的语法中。第一步（第 1 行）是对下标表达式进行求值。由于在 Modula-2 和 Pascal 语言中，数组元素可从任意下标值开始，所以下一步（第 2~4 行）用于减去数组第一维下标的下界。对于一个声明为 [3 .. 9] 的数组而言，一个计算结果为 6 的下标值必须访问数组中的第 4 个元素，而不是第 6 个元素。C 语言强制规定所有下标范围的下界均为 0，因而无需这一步骤。使用简单的常量表达式求值优化技术，即可将这一步骤转移到编译时执行，从而它不会成为运行时的负担。应根据数组中的元素个数对偏移量下标表达式执行范围检查（第 5~15 行），又或者参照下标标量类型的上界（和下界）对原下标表达式进行范围检查。接着必须乘以分量的大小，从而得到该下标所确定的分量的偏移量（第 16~17 行）。最后，偏移量累加到正在处理的 L 表达式的值（第 18 行），并派生出一个新的待处理 L 表达式。

一个多维数组等价于多个数组的一维数组；实际上，在 Modula-2 语言中显式地将一个多维数组表达成这样。如果在语言定义中二维数组和一个数组的数组互不相同，则符号表中的定义可添加一个标志位表示它属于哪一种；否则，下标表达式中的逗号可当作“][”一样处理其语义，类型指示符中的逗号可当作单词“OF ARRAY”一样处理。

6.10 其他数据结构

大多数其他的数据结构都是上述三种基本方案的变形。Pascal 语言有一个文件类型（但 Modula-2 语言中没有），然而其语法和语义均与指针类型相同。一个文件变量可能比一个普通指针略大，从而隐藏的字段可引用适当的操作系统钩子函数以实现读写，并且还可能提供了数据的缓冲。

Modula-2 语言允许声明 **PROCEDURE** 类型的变量。一个过程变量可被赋值，可作为参数传递给另一过程，也可直接将它应用到一个参数表以调用它。过程变量必须足够大，才足以保存过程入口点的地址以及它的当前父帧指针。由于 Modula-2 语言仅允许将外层过程赋值给变量或作为参数传递，所以帧指针是不必要的：它始终采用全局的帧指针。其他语言可能要求一个过程变量同时携带帧指针信息；在这种情况下，如果一个帧尚有未关闭的引用，则该帧不可以被回收空间。通常认为管理这种状况会因过于复杂而得不偿失。

Modula-2 和 Pascal 语言均支持集合类型 **set**，并实现为一种布尔值的压缩数组。大多数计算机硬件都提供了指令实现单个字的交集与并集(**And** 和 **Or**)。如果允许集合类型大于 Modula-2 语言中占 1 个字的 **BitSet** 类型，那么编译程序设计人员就必须设计合适的编程方法以实现多个字的逻辑运算。由于这一工作除了有助于正在讨论的数据结构之外，对领悟编译程序的设计没有什么特别启示，并且在 Modula-2 和 Pascal 源程序中集合运算符通过重载算术运算符“*”和“+”表示（导致不可能采用语法制导代码生成技术），故本书不再赘述这些内容。

6.11 Itty Bitty 栈机器的输入和输出

作为一种教学装置，Itty Bitty 栈机器没有输入和输出方面的需求，因为已可通过检查程序执行的踪迹来校验小型测试程序的正确运行。然而这对于大一些的程序却会造成不便。

真正的计算机通常按以下两种方式之一处理输入和输出：一些计算机有专门的 I/O（输入和输出）地址空间，并有专门的指令将数据传送到其中；另一些计算机没有专门的 I/O 操作，但指派了一部分主存空间作为 I/O 硬件传送的寄存器，称之为内存映射 I/O。当然，只要设计人员愿意选用，任何计算机都可使用内存映射 I/O，包括那些使用 I/O 指令的计算机。

IBSM 被设计为模拟内存映射 I/O；在所模拟的内存地址空间之外，物理内存地址-1 是一个单字符的 I/O 端口。存入该位置的数字将被转换为单个字符，然后被传送到控制台终端或标准输出；将该地址中的数据取到栈中相当于从终端键盘或标准输入文件中读入单个字符，并将其序数压入 IBSM 的运行栈中。

与大多数计算机一样，将多个整数转换为字符串需要一个库过程。但这并非本章的重点，这一过程的实现留作练习。

6.12 语法制导语义的局限

本章已数次遭遇“语法制导语义能够做什么”这一问题。第5章深入讨论了如何基于符号表中存储的标识符相关信息实现类型检查，采用这种方式能够处理两类语义信息。首先，须根据语法形式进行校验的所有信息必须组织为一种独立于标识符语义类别的形式；因而，我们为所有标识符都利用个位数存储了其“类型”信息，而不管这些标识符到底是变量、类型还是函数。类似地，对所有标识符也将其“类别”信息记录在十位数中。

在符号表中，语法制导语义可处理的第二类信息片段特定于某些标识符类别，但仅限于可在语法上惟一确定其类别的标识符。类型名字可在语法上确定，譬如出现在 **TYPE** 语句中，或者出现在 **VAR** 和函数声明中的冒号右边；这些位置中出现的任何标识符必定是一个类型名字。采用类似方法，可从语法上识别一个变量名字，譬如出现在赋值语句的左边，或者出现在表达式中但后面没有跟着圆括号。

不幸的是在实际程序设计语言中，标识符可不带圆括号出现在表达式中，但它们并不是变量；它们可以是已声明的常量，在 Pascal 语言中它们还可以是函数调用（但 Modula-2 并非如此）。在 Modula-2 语言中，不带圆括号出现在表达式中的函数名字是一个常量（其类型为 **PROCEDURE**），因而在形式上无法与其他常量区别开来。

利用一个属性文法的语法制导语义，在处理变量的同时对常量和函数调用（Pascal 语言中）执行充分的类型检查并不困难；如前所述，这只要求所有类别的标识符的类型信息必须得一致地放置在符号表的值记录中即可。然而在代码生成的情况下，这种一致性不复存在。在 IBSM 模型的体系结构中，一个局部变量需要生成 3 个字的代码：一条 **LoadCon** 指令装入变量地址，跟着一条 **Load** 指令取出该变量的值；而常量只需要生成一条 **Load** 指令装入该常量的值。如果没有语法上的区别可驱动对语义的不同处理，就无法正确地生成两种不同类别的代码。

如果我们允许非局部变量，即使在变量之中也会冒出类似的问题。利用 3 个字的简单指令序列即可访问一个局部变量，但一个非局部变量则需要更多或更少的代码，取决于该变量的帧

离得多远。我们回避这一问题的办法是强制所有的变量引用都必须通过 `Display` 表，以牺牲一些代码效率为代价。这特别值得注意，因为大多数过程仅访问局部变量，而我们却强制所有过程都要构建一个高成本的 `Display` 表，并通过它传递所有的变量引用。第 8 章将讨论在一个属性文法中如何设计语义制导代码生成的方法。

6.13 手工编写编译程序的代码生成

正如第 5 章所述，“一遍”递归下降分析程序在结构上与属性文法稍有区别。在一个输出文件中产生输出代码的字符串（用方括号中带引号的字符串表示）只不过是在 `Modula-2` 语言的代码中适当地调用 `WriteString` 过程（Pascal 语言中的 `write`）而已。内置属性求值函数 `number` 和 `newline` 很容易对应到标准过程 `WriteInt` 和 `WriteLn`（Pascal 语言中的 `write` 和 `writeln`）。如果代码被缓存在一个数组的数据结构中，地址就对应到数组的下标，代码只不过是赋值给合适的元素。

如果目标机器具有比 `IBSM` 更复杂的指令集，则可利用大多数语言支持的位移库例程将指令字中的分量字段组合在一起，或者像本书之前压缩符号表的值那样利用乘法和加法。

6.14 语法制导语义的应用

编译程序的设计原则可直接应用于与形式化定义的语言有关的各种各样的编程任务。大多数程序要求在执行动作之前，先检查用户输入在语法以及语义上的一致性。一致性检查相当于一个编译程序或解释程序的前端，执行动作则相当于后端。

本节分析两个不同的应用，展示了语法制导的语义如何使得这类应用更易于实现。其中的一个应用是程序设计语言的解释程序，它在大多数方面类似于一个编译程序，只是其后端的语义动作是立即运行程序，而不是生成代码以后再运行。另一个应用是一个美化打印工具，它根据语法驱动的提示信息决定换行和缩进的位置。

6.14.1 Tiny BASIC 解释程序

为演示一个程序设计语言解释程序的基本性质，我们关注一个没有静态语义的语言：只有整数的 `Tiny BASIC` 语言。`Tiny BASIC` 在早期的微机上较为流行，它有 7 种语句类型和 26 个预声明的变量（每个变量用一个字母表示）。代码清单 6.4 给出的文法略有简化。

代码清单 6.4 `Tiny BASIC` 语言的语法规则，包括非形式化的语义动作

<code>Cmd</code>	
<code>-> Stmt</code>	[只需执行]
<code>-> NUM textLine</code>	[以行号次序插入内存中]
<code>-> NUM</code>	[从内存中删除该编号指定的行]
<code>-> "CLEAR"</code>	[从内存中清除程序]
<code>-> "RUN"</code>	[在内存中找到第一行，并开始执行]
<code>Stmt</code>	
<code>-> "LET" VAR "=" Expn</code>	[从表达式栈中弹出，并存放到变量中]
<code>-> "IF" Expn ("=" "<" ">") Expn "THEN" Stmt</code>	[弹出两个值并作比较；仅当为真时执行该语句]
<code>-> "GOTO" Expn</code>	[从表达式栈中弹出一个值；从该值表示的行继续执行]
<code>-> "INPUT" VAR</code>	[从终端接受一个数字，并存入变量中]

```

-> "PRINT"  Expn          [弹出表达式,并在终端上显示]
->                                     [空语句,什么也不做]

Expn
-> Term  (( "+" | "-" ) Term [弹出两个值;执行加法或减法;将结果压入栈中] ) *

Term
-> Fact  (( "*" | "/" ) Fact [弹出两个值;执行乘法或除法;将结果压入栈中] ) *

Fact
-> VAR          [将变量的值压入栈中]
-> NUM          [将数字的值压入栈中]
-> "("  Expn  ")"

```

实现该解释程序最困难的部分是在内存中维护程序代码。一种简单的实现是使用一个定长字符串的数组,以行号作为下标。在程序启动时,或执行一条 **CLEAR** 命令之后,所有字符串均为空(有欠优雅的做法是用空格填充这些字符串)。如果用户输入一个行号,则以输入行中余下的所有内容填充该行号指定的行。最早的微机版 Tiny BASIC 实现只有非常有限的内存:一些甚至运行在只有 2,048 字节这么少的内存中,包括解释程序、BASIC 程序以及所有的数据空间。BASIC 程序存储在一个字符数组中,同时存储的还有单独以双字节二进制数编码的行号。在 Modula-2 语言的实现中,一种合理的折中方案可能引入一个按行号次序维护的、按行存放的数组;在这个按行存放的数组中,每一入口可包含一个行号和一个指向字符数组的索引。这里不需要长度信息,因为任一行的长度正是该行的起始索引与下一行的起始索引之差。

借助于运行时的表达式栈,表达式求值相当简单。每次引用一个变量或常量时,将该变量或常量的值压入栈中。每一运算符从栈中弹出其操作数,然后执行这两个值的运算,最后将运算结果压入栈中。**IF** 语句先对比较运算进行求值,然后若结果为假则跳过语句的其余部分。**GOTO** 语句对一个表达式进行求值,然后找出行号与求值结果相同的那一行(如果该行不存在则报告一个错误),然后从这一行开始继续执行。这些实现留作练习。

6.14.2 Micro-Modula 美化打印工具

尽管本例仅展示了将缩进后的代码清单输出为文本文件,但美化打印工具的另一常见用法是一个语法制导的文本编辑程序,其中语言的文法控制着屏幕的显示。允许动态重构数据项的显示方式会引入太多的复杂性,已超出我们在这里的分析范围。同样道理,我们仅探讨几个控制结构,每一控制结构都展示了格式化过程中的不同特点。代码清单 6.5 是这一美化打印工具的属性文法的核心部分。

代码清单 6.5 Micro-Modula 语言源程序的美化打印

```

declns ↓indent

-> "PROCEDURE" ID ↑name ";"
    newline ↓indent ["PROCEDURE "; spell ↓name; ";"]
    (declns ↓indent+1)*
    "BEGIN"
        newline ↓indent ["BEGIN"]
    (stmts ↓indent+1)*
    "END" ID ↑name ";"
        newline ↓indent ["END "; spell ↓name; ";"]

```



```

->  "VAR" ID ↑name ":"      newline ↓indent ["VAR "; spell ↓name; ":"]
    typeden ↓indent ";"    [";"]
    ;

stmts ↓indent

->  "IF"                      newline ↓indent ["IF "]
    expn ↓indent+1
    "THEN"                    [" THEN"]
    (stmts ↓indent+1)*
    ("ELSE"                   newline ↓indent ["ELSE"]
    (stmts ↓indent+1)*
    )?
    "END"                     newline ↓indent ["END"]
    (";"                      [";"]
    stmts ↓indent
    )?

->  ID ↑name ":@"           newline ↓indent [spell ↓name; " := "]
    expn ↓indent+1
    (";"                      [";"]
    stmts ↓indent
    )? ;

expn ↓indent

->  factor ↓indent+1
    (operator
    factor ↓indent+1
    )* ;

factor ↓indent

->  ID ↑name                  [spell ↓name]

->  NUM ↑value                [number ↓value]

->  "("                       ["("]
    expn ↓indent+1
    ")"                       [")"]
    ;

operator

->  "+"                       ["+"]
    | " *"                    ["*"]
    | "="                     ["="]
    | "<"                      ["<"]
    ;

newline ↓indent              { 开始新的一行, 然后输出空格 }

```

```

->                                [startline]
    space ↓indent ;

space ↓indent                    { 若 indent 为 0 则无空格 }

->                                [indent > 0; " "]
    space ↓indent-1

->                                [indent = 0]
    ;

```

代码清单 6.5 中的文法未给出表达式中运算符的优先级，因为这些语义动作只是关心在某一位有一个运算符，而不关心它是一个什么运算符。该文法也没有涉及一行的长度。一个更完善的美化打印工具会携带一个从左到右传递的属性，用于记录当前行的剩余空位；如果表示缩进值的继承属性小于某一阈值，则以适当的缩进（向前进 1 步或 2 步）开始一个新行，否则当下一单词在当前行没有足够的空位时，跳回到左边界附近。

该文法最后的产生式引入一个约束，这一约束在它的第一条产生式不一定为真，即指定的缩进值要么是 0（作为递归的基数时），要么大于 0（从左边界起填充空格时）。支持这一决策的代码相当于在普通 LL(1) 文法中测试一个向前看符号；但在这种情况下，非终结符包含一个产生空串的选项。第 8 章将深入探讨这类语义驱动决策的概念。

小结

对源程序文本的编译就是将它翻译为某一目标机器上的机器语言，这一机器语言程序称为目标代码。一种翻译方法是在源语言文法的重写规则中定义代码生成序列，该技术称为语法制导代码生成。当应用一条重写规则时，编译程序调用相应的代码生成例程，从而创建目标代码。使用属性文法有可能生成部分依赖于语义信息的目标代码。

本章介绍了一个虚构的计算机 IBSM（Itty Bitty 栈机器）用于演示代码生成。为演示这些技术，我们为第 5 章介绍的 Micro-Modula 语言的文法（参阅代码清单 5.2）扩展了新的代码生成语义。

关键术语

accumulator（累加器） 一个寄存器，用于存储算术运算或逻辑运算的数据。

backpatching（回填） 在一个编译程序中，在编译的时候保存每一不完整的向前分支记录；当目标地址变为已知时，回到已生成的输出代码，并将正确的值插入分支地址中。

compiler（编译程序） 一个分析程序，并且还将源代码文本翻译为目标代码。

back end（后端） 生成目标代码。

front end（前端） 识别一个语法上正确的源程序文本。

two-pass（两遍） 编译程序两遍扫描源程序，以解析向前引用。

display（Display 表） 帧指针的向量（数组）。

goto instruction（GOTO 指令）

（1）一条分支指令。

（2）一条非结构化（无条件）跳转语句。

I/O（输入/输出） 计算机输入和输出的常见缩写。

lex level（词法层次） 在静态作用域语言中，源程序文本中某一特定位置的外层包围过程的数目，在编译时即可确定。

memory-mapped (内存映射的) 计算机 I/O 的一种实现方式, 将一部分内存地址空间分配为输入和输出寄存器。

object code (目标代码) 目标机器的语言。

orthogonality (正交性) 指计算机体系结构的正交性, 是一条指令的寻址模式独立于操作的程度。如果每一指令具有相同的寻址模式, 则称该机器为完全正交的。

register (寄存器)

(1) 高速数据存储电路。

(2) 计算机中单独可编址的一类存储空间, 访问时比主存更简单、更快速。

syntax-directed code generation (语法制导代码生成) 代码生成序列定义在语言的语法规则中, 而不是根据其静态语义来选择。

练习

- 为以下每一小题编写 Itty Bitty 栈机器的“汇编语言”代码 (使用助记符, 并加上适当的注释), 并以手工方式汇编为机器代码。
 - 给定栈顶两个数, 弹出 (并丢弃) 其中较小的数, 保留较大的数。
 - 求 1 到 n 的连续奇数之和, 其中 n 由内存中的一个变量指定。
 - 给定存放在内存中的某一变量中的任意数, 计算其平方根: 方法是计算在它变为负数之前可减去的连续奇数的个数。提示: $1 + 3 = 4 = 2^2$, $1 + 3 + 5 = 9 = 3^2$, 如此类推。
 - 在 IBSM 中没有单条指令将栈顶的布尔值反转, 即将 **false(0)** 改为 **true(1)**, 反之亦然。试给出两个不同的 IBSM 指令序列实现这一功能。
 - 给定栈中两个 (局部) 变量地址和一个数 $n > 0$, 将 n 个连续的字从一个地址搬迁到另一地址中, 并在完成后从栈中删除这 3 个值。仅使用栈作为临时存储空间, 不要使用内存中的任何辅助变量。这一代码可用在编译程序中实现一个数据结构的赋值。
 - 在 IBSM 中没有指令执行一个数与另一个数的除法。编写一个过程, 它接受两个参数 **a** 和 **b**, 返回 **a DIV b** 的商。除数 **b** 为 0 时停机, 但对负数 **a** 或 **b** 应给出正确的结果。一种简单 (但费时) 的方法是计算从被除数可减去除数的次数。计算 **a DIV b** 的更快例程是以下算法:

```

n := 1;
WHILE b <= a DO
    b := b + b;
    n := n + 1
END;
q := 0;
WHILE n > 0 DO
    IF a < b THEN
        q := q + 1;
    ELSE
        a := a - b;
        q := q + n + 1
    END;
    a := a + a;
    n := n - 1
END;
RETURN q;

```

- 修改练习 1 (f) 的代码, 接受负数作为被除数和 (或) 除数, 并给出正确的有符号结果。

- 以某种高级语言 (如 Modula-2 语言) 编写一对过程, 在一个字符 I/O 端口上实现整数的 I/O。

- (a) 输入功能应循环地读入一个特定的全局（整型）变量，并将这些 ASCII 数位组装为一个整数后，作为函数结果返回。
- (b) 输出功能应将其参数分解为每次处理一个字符，将这些字符存储到同一全局变量中；不可使用数组作为中间存储。
- (c) 将你编写的过程翻译为 IBSM 代码；使用以下 IBSM 操作序列访问字符端口：

```
LoadCon 1
Negate
Global
<load 或 store>
```

- 3. 编写一个属性文法，为一个 **WHILE** 循环生成正确的 IBSM 代码。
- 4. (a) 为 Tiny BASIC 的语义编写一个形式化的属性文法。
- (b) 用 Modula-2 或 Pascal 语言为 Tiny BASIC 编写一个递归下降解释程序。

复习小测验

指出下列陈述是否正确。

- 1. 语法制导代码生成意味着代码生成序列由一个描述了可接受的源程序文本的语法定义，而不是由嵌套在文法中的语义分析定义。
- 2. 一个编译程序的后端可专门只负责完成代码生成工作。
- 3. 在 IBSM 指令集中，操作 **zero** 表示清空内存。
- 4. 使用语法制导代码生成技术扩展 Micro-Modula 语言，从而可正确地编译常量标识符。
- 5. 不可能生成测试一个条件并在条件为真时分支的 IBSM 代码。

编译程序实验项目

- 1. 为你的 Itty Bitty Modula 语言编译程序的文法：
 - (a) 添加必要的语义以生成 IBSM 代码。在初次尝试时，先不考虑对过程的处理。
 - (b) 在 TAG 编译程序中编译你的编译程序，或者以手工编码方式将新的变化添加到你的递归下降分析程序中。
 - (c) 通过编译一些小型的 IBSM 程序测试你的编译程序，并在 IBSM 上运行它们。注意确保嵌套循环和条件语句可以正确地运行。
- 2. 为你的文法添加语法以及正确的语义，以实现：
 - (a) 子界类型（含范围检查）和 **CHAR** 类型。
 - (b) 记录类型（但不要打算实现 **WITH** 语句）。
 - (c) 指针类型。
 - (d) 数组类型（这需要子界类型）。
 - (e) **FOR** 循环。
 - (f) 无参数的函数（但不要打算实现过程变量）。
 - (g) 最多只有两个参数的过程（不要打算实现变参）。
 - (h) 内置库过程 **ReadChar** 和 **WriteChar**（使用保留字作为过程名，并使用内联代码）。

进一步阅读

Cattell, R.G.G. "Automatic Derivation of Code Generators from Machine Description." *ACM Transactions on Programming Languages and Systems*, Vol.2, No.2 (April 1980), pp.173-190.

所给出的方法基于一种所谓的树产生式的模板形式，这些产生式一起被收集到机器表中；在代码生成时，使用一种模式匹配技术，分析树中的结点与机器表中对应的产生式匹配（参阅文中第 3 节对该方法的描述）。

Ganapathi, M. *Retargetable Code Generation and Optimization Using Attribute Grammars*, Ph.D. Dissertation, University of Wisconsin at Madison, 1980.

将 Graham-Glanville 方法扩展为一种完整的属性文法方法学（参阅 D. Spector 的综述文章）。

Ganapathi, M. & Fischer, C.N. "Affix Grammar Driven Code Generation." *ACM Transactions on Programming Languages and Systems*, Vol.7, No.4 (October 1985), pp.560-599.

参阅第 1 节，特别是第 562~563 页，其中总结了附加文法策略。

Hopgood, F.R.A. *Compiling Techniques*, New York: American Elsevier, 1969.

参阅第 8 章“算术表达式的代码生成”，其中讨论了针对一台虚构的单累加器计算机的代码生成。

Jacobi, C. *Code Generation and the Lilith Architecture*, Ph.D. Dissertation, Swiss Federal Institute of Technology, 1951.

Lilith 计算机是一种面向 Modula-2 指令集设计的栈机器，参阅其中第 4 章关于编译问题的讨论。

Lukasiewicz, J. "Formalization of Mathematical Theories." Paris, 1953. In *Jan Lukasiewicz: Selected Works* ed. L. Borkowski. Amsterdam, Netherlands: North-Holland, 1970.

参阅第 1 节，卢卡西维茨介绍了数论中的无括号符号化方法。

Lukasiewicz, J. *Elements of Mathematical Logic*. Oxford, Eng.: Pergamon Press, 1963.

参阅第 I.2 部分，卢卡西维茨介绍了他为命题逻辑开发的无括号符号化方法中的表示法。

Spector, D. & Turner, P.K. "Limitations of Graham-Glanville Style Code Generation." *ACM SIGPLAN Notices*, Vol.22, No.2 (February 1987), pp.100-108.

参阅第 3.3 节，介绍了一种所谓的“上一下”分析方法（使用一种完全由表格驱动的分析算法）；这种分析文法不同于 Graham-Glanville 采用的从左到右分析方法。

$$\begin{array}{ll} E \rightarrow TS & P \rightarrow *FP \\ S \rightarrow +TS & P \rightarrow \epsilon \\ S \rightarrow \epsilon & F \rightarrow (E) \end{array}$$

$T \rightarrow FP$ $F \rightarrow n$

表 7-1 自顶向下 PDA 的栈踪迹展示了最左推导过程；第 T.11 步以空心字体表示的串是推导到该步骤时的一个句型

步 骤	已读入的输入串	未读的输入串 / 栈中的内容	产生式
T.0	无之前的输入	. $n + n * n$ 未读的输入 . E 栈	
T.1		. $n + n * n$ 未读的输入 . TS 栈	$E \rightarrow TS$
T.2		. $n + n * n$ 未读的输入 . FPS 栈	$T \rightarrow FP$
T.3		. $n + n * n$ 未读的输入 . nPS 栈	$F \rightarrow n$
T.4	之前的输入 n	. $+ n * n$ 未读的输入 . PS 栈	(读输入符号)
T.5	之前的输入 n	. $+ n * n$ 未读的输入 . S 栈	$P \rightarrow \epsilon$
T.6	之前的输入 n	. $+ n * n$ 未读的输入 . +TS 栈	$S \rightarrow +TS$
T.7	之前的输入 $n +$. $n * n$ 未读的输入 . TS 栈	(读输入符号)
T.8	之前的输入 $n +$. $n * n$ 未读的输入 . FPS 栈	$T \rightarrow FP$
T.9	之前的输入 $n +$. $n * n$ 未读的输入 . nPS 栈	$F \rightarrow n$
T.10	之前的输入 $n + n$. $* n$ 未读的输入 . PS 栈	(读输入符号)
T.11	之前的输入 $n + n$. $* n$ 未读的输入 . *FPS 栈	$P \rightarrow *FP$
T.12	之前的输入 $n + n *$. n 未读的输入 . FPS 栈	(读输入符号)
T.13	之前的输入 $n + n *$. n 未读的输入 . nPS 栈	$F \rightarrow n$
T.14	之前的输入 $n + n * n$. 未读的输入 . PS 栈	(读输入符号)
T.15	之前的输入 $n + n * n$. 未读的输入 . S 栈	$P \rightarrow \epsilon$
T.16	之前的输入 $n + n * n$. 未读的输入 空栈	$S \rightarrow \epsilon$

表 7-1 展示了一个自顶向下分析程序在分析输入串 $n+n*n$ 时的踪迹。注意, 如果将之前读入的输入串与栈的内容一起好像放在同一行那样阅读 (如表中第 T.11 步所示), 结果将是一个句型。表 7-2 展示了以同一文法的自底向上分析程序的 PDA 分析同一个串的过程, 然而这里的栈位于左边, 将栈中的内容与仍未读入的输入串好像在同一行中那样阅读, 结果是一个句型 (如表中第 B.7 步所示)。

表 7-2 自底向上 PDA 的栈踪迹 (自底向上读取踪迹) 展示了最右推导过程:

第 B.7 步以空心字体表示的串是推导到该步骤时的一个句型

步 骤	已读入的输入串 / 栈中的内容	未读的输入串	产生式
B.0	无之前的输入 空栈	$n+n*n$ 未读的输入	
B.1	之前的输入 $n.$ 栈 $n.$	$+n*n$ 未读的输入	(读输入符号)
B.2	之前的输入 $n.$ 栈 $F.$	$+n*n$ 未读的输入	$F \rightarrow n$
B.3	之前的输入 $n.$ 栈 $FP.$	$+n*n$ 未读的输入	$P \rightarrow \epsilon$
B.4	之前的输入 $n.$ 栈 $T.$	$+n*n$ 未读的输入	$T \rightarrow FP$
B.5	之前的输入 $n+.$ 栈 $T+.$	$n*n$ 未读的输入	(读输入符号)
B.6	之前的输入 $n+n.$ 栈 $T+n.$	$*n$ 未读的输入	(读输入符号)
B.7	之前的输入 $n+n.$ 栈 $T+F.$	$*n$ 未读的输入	$F \rightarrow n$
B.8	之前的输入 $n+n*.$ 栈 $T+F*.$	n 未读的输入	(读输入符号)
B.9	之前的输入 $n+n*n.$ 栈 $T+F*n.$		(读输入符号)
B.10	之前的输入 $n+n*n.$ 栈 $T+F*F.$		$F \rightarrow n$
B.11	之前的输入 $n+n*n.$ 栈 $T+F*FP.$		$P \rightarrow \epsilon$
B.12	之前的输入 $n+n*n.$ 栈 $T+FP.$		$P \rightarrow *FP$
B.13	之前的输入 $n+n*n.$ 栈 $T+T.$		$T \rightarrow FP$

(续)

步 骤	已读入的输入串 / 栈中的内容	未读的输入串	产生式
B.14	之前的输入 $n + n * n .$		$S \rightarrow \epsilon$
	栈 $T + T S .$		
B.15	之前的输入 $n + n * n .$		$S \rightarrow + T S$
	栈 $T S .$		
B.16	之前的输入 $n + n * n .$		$E \rightarrow T S$
	栈 $E .$		

自顶向下分析程序有时也称预测分析程序，因为（除向前看符号之外）它在读入一个输入符号之前会预测输入串中该符号会是什么。自底向上分析程序将输入串读入栈中，一旦在栈的顶部符号中识别出产生式的右部就立即应用该产生式。

对于这里的文法 G_{27} 例子，在分析过程中通常同时有多个机会将某一产生式应用到栈顶的符号。例如，表 7-2 中第 B.14 行选择了应用产生式 $S \rightarrow + T S$ ，尽管我们也可能轻易地选择了 $S \rightarrow T S$ 甚至 $S \rightarrow \epsilon$ 。类似于自顶向下分析，多条可用产生式之间的选择在必要时须借助于向前看符号，又或从输入串中读入另一符号到栈中，而不是应用一条产生式。然而，决定应用哪一条产生式所需的大部分信息往往包含在分析程序的栈中。直到产生式右部的所有单词已读入并表示在栈顶，才决定应用哪一条产生式，这样就能保证如果基于 k 个向前看符号有可能实现确定的分析，那么就可以构建一个 $LR(k)$ 分析程序。高德纳证明了一个有穷状态自动机 (FSA) 可检查栈中的内容，使得一个不超过 k 个向前看符号的自底向上分析程序的 PDA 可以确定地选择应用正确的产生式（如果存在这样的产生式）。这一 FSA 是所有 $LR(k)$ 分析程序的基础。栈可以有任意的深度，因而分析程序的每次状态变迁都检查栈中的全部内容将略显冗长，幸好这并不是必要的。

7.2 LR(k)分析程序

考虑一个简单文法 G_{41} ：

$$\begin{aligned} S &\rightarrow a A d & A &\rightarrow c \\ S &\rightarrow b B d & B &\rightarrow c \end{aligned}$$

该文法是 LL(1)的，故显然可构建一个确定的分析程序。以自底向上方式分析输入串 bcd 时，在将两个单词读入栈中后，向前看符号是 d ，这对于选择应用哪一条产生式毫无帮助。如果分析程序错误地选择了产生式 $A \rightarrow c$ ，那么在读入 d 后将阻塞，因为没有产生式容许串 bAd ，如表 7-3 所示。然而，通过分析栈中内容可知，惟一正确的选择必定是产生式 $B \rightarrow c$ 。

表 7-3 在忽略栈中内容的情况下，文法 G_{41} 的自底向上分析的尝试踪迹

之前的输入	$b c d$	未读的输入	
空栈			
之前的输入 $b .$	$c d$	未读的输入	(读输入符号)
栈 $b .$			

(续)

之前的输入	$b c .$	d	未读的输入	(读输入符号)
栈	$b c .$			
之前的输入	$b c .$	d	未读的输入	$A \rightarrow c \text{ ??}$
栈	$b A .$			
之前的输入	$b c d .$			(读输入符号)
栈	$b A d .$			
之前的输入	$b c d .$			(阻塞)
栈	$b A d .$			

在任一决策点，仅存在有穷数目的可能选择，因而如果本来就存在一个确定的分析过程，就可用一个有穷状态自动机遍历并分析栈中的内容，最终得出决策结果。文法 G_{41} 的这个 FSA 如图 7-2 所示。留意文法 G_{41} 是 LR(0) 的，亦即无需任何向前看符号即可确定地分析其语言中的所有串。当然，该语言还是有穷的，并且没多大意思。

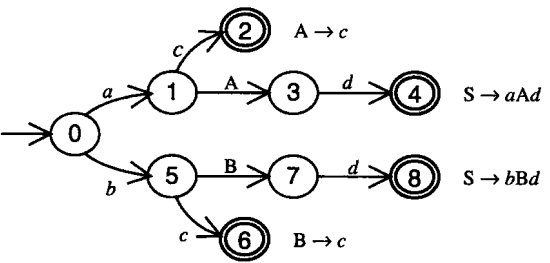


图 7-2 文法 G_{41} 的 LR(0) 栈的有穷状态自动机

该 FSA 按如下方式工作。开始时状态 0 位于栈的底部，栈中每一符号均前进一个状态，直至到达栈的顶部。如果栈顶是一个停机状态，则应用该状态所指示的产生式，即从栈中弹出该产生式右部的所有符号，然后以产生式左部的非终结符取而代之；如果栈顶到达的是任何其他状态，则读入下一输入单词并压入栈中。在这两种情况下，FSA 均从栈底的状态 0 重新启动。如果 FSA 未到达顶部即阻塞，则报告一个错误：输入串不属于该语言。

7.2.1 构造 LR(k) 状态机

控制一个 LR(k) 分析程序的有穷状态自动机可根据语言的文法以算法方式自动地构造。这一构造过程复杂且冗长，故而是一个真正的编译程序不会以手工方式构造。

文法 G 的语言分析程序的 FSA 中，每一状态 q 由 PDA 格局的一个等价类（即“项目”）组成。一个项目写作如下形式：

$A \rightarrow u \bullet v ; \sigma$

其中， $A \rightarrow uv$ 是文法 G 中的一条产生式， σ 是一个由 k 个终结符组成的串。分隔符“ \bullet ”在某种意义上可理解为 FSA 在遍历栈中内容时的读磁头；尽管这样有助于直观理解，但这样的理解却是不够严谨的，因为栈中实际上包含了一个句型的全部左部，而项目仅对单条产生式进行编码。

一个项目在其右部包含一个分隔符， u 和 v 分别表示在这一特定项目中分隔符左边和右边的、由终结符与非终结符组成的串（可能为空串）。根据这一上下文中的非终结符的 $Follow_k$ 集

可推导出串 σ , σ 将用于确定向前看符号; 我们很快将看到, 不同类的 LR 文法在很大程度上就是由 σ 的计算方式决定的。

FSA 的起始状态 q_0 初始时是所有项目的集合:

$$S \rightarrow \bullet w; \perp$$

其中, S 是文法 G 的目标符号, $S \rightarrow w$ 是一条产生式, “ \perp ” 是一个表示输入结束的单词 (正如之前的假设, 其长度不超过 k 个符号)。对于文法 G_{41} , q_0 的初始项目集是以下集合:

$$S \rightarrow \bullet a A d; \perp$$

$$S \rightarrow \bullet b B d; \perp$$

从起始状态 q_0 出发, 每次定义一个新状态时, 调用状态上的 *closure* 运算, 即对每一项目:

$$A \rightarrow u \bullet B v; w$$

以及 $First_k(vw)$ 中的每一个元素 y , 在项目集中添加新的项目:

$$B \rightarrow \bullet x; y$$

其中, $B \rightarrow x$ 是文法 G 中的一条产生式。

闭包运算 *closure* 不会为文法 G_{41} 的起始状态 q_0 添加新项目。一个项目集表示了分析程序自动机中的一个状态, 闭包运算记录了该文法中可在一个推导过程中应用的所有可能的产生式, 这一分析过程将在 PDA 中传递此状态。直观地看, 这在概念上类似于将构造一个非确定的 FSA 看作从正则文法转换为一个 FSA 的构造过程的一部分, 尽管这两者并非完全相同。实质上, 闭包运算展开了一个项目集的所有项目中直接跟在分隔符右边的非终结符, 反映了在分析语言中一个串的过程中该产生式的可能应用。

从一个状态转入另一状态的变迁被定义为在一个项目中将分隔符向右前进一个符号 (终结符或非终结符)。因而, 给定一个非终结符或终结符 x , 状态 q_i 中每一项目:

$$A \rightarrow u \bullet x v; y$$

将在新状态 q_j 中创建一个新项目:

$$A \rightarrow u x \bullet v; y$$

并为 FSA 的变迁集添加一条新变迁:

$$\delta(q_i, x) = q_j$$

只要存在一条沿一个终结符或非终结符 x 出发的变迁, 就构建这一新的状态; 从某一特定状态 q_i 沿标记为 x 的所有变迁出发, 都会到达相同的状态 q_j 。在文法 G_{41} 的例子中, 基于 q_0 的项目集中的第一个项目, 可构造从 q_0 到 q_1 的变迁 a ; q_0 中的第二个项目产生一条到达另一不同状态 (譬如 q_5) 的变迁。

如果闭包运算得到的一个新状态 q_j 与某一现有的状态 q_h 具有相同的项目列表, 则丢弃新状态, 并在每一新变迁中以 q_h 取代 q_j 。注意, 如果由闭包运算添加到一个状态中的所有项目被作上标记, 且新状态与另一状态中的所有无标记项目相同, 则在调用闭包运算之前即可确定重复出现的状态。无标记的项目集有时又称为“核”。须小心检查这些状态以保证完全相等, 因为一个新状态在创建时可能比其他状态多一些或少一些项目; 这些状态不会被看作是相同的状态。

由于仅含有穷数目的产生式, 每一产生式中仅有有穷数目的分隔符位置, 且长度为 k 的所有串的数目也是有穷的, 由上述过程创建的状态数目自然也是有穷的, 因此算法必定是可终止的。

7.2.2 一个 LR(2)分析程序

让我们构造一个 LR(2)分析程序的 FSA 识别上述文法 G_{41} 的语言。初始状态 q_0 中有两个项目：

$S \rightarrow \bullet a A d; \perp$

$S \rightarrow \bullet b B d; \perp$

如前所述，由于不存在一个项目中有非终结符直接跟在分隔符之后，所以闭包运算不会添加任何项目。现在添加变迁 $\delta(q_0, a) = q_1$ 后，创建一个新状态 q_1 ：

$S \rightarrow a \bullet A d; \perp$

该状态开始时仅有一个项目，因为 q_0 中只有一个项目在其分隔符的右边有 a 。由于这一项目中有一个非终结符 A 位于分隔符的右边，闭包运算根据产生式 $A \rightarrow c$ 添加一个新的项目：

$A \rightarrow \bullet c; d \perp$

新的 Follow 串通过以下方法构造：在那些能跟随非终结符 A 的串（即单词 d ）的右端再拼接上之前的 Follow 串（ \perp ）。由于现在构造的是 LR(2)识别程序，我们需要的是上述结果的 $First_2$ 集合，即“ $d\perp$ ”整个串。

添加变迁 $\delta(q_1, c) = q_2$ 后，构造了状态 q_2 （闭包运算没有添加任何项目）：

$A \rightarrow c \bullet; d \perp$

从 q_1 中其他可能的变迁出发，可类似地构造状态 q_3 和 q_4 ；根据 q_0 中的第二个项目，可类似地构造状态 $q_5 \sim q_8$ 。留意状态的名字是相当随意的，我们直接按创建的次序为这些状态编号。表 7-4 展示了构造过程完成后的结果，与图 7-2 所示的 FSA 相同。

表 7-4 文法 G_{41} 的 LR(2)分析程序的状态

状 态	项 目	变 迁
q_0	$S \rightarrow \bullet a A d; \perp$	$\delta(q_0, a) = q_1$
	$S \rightarrow \bullet b B d; \perp$	$\delta(q_0, b) = q_5$
q_1	$S \rightarrow a \bullet A d; \perp$	$\delta(q_1, A) = q_3$
	$A \rightarrow \bullet c; d \perp$	$\delta(q_1, c) = q_2$
q_2	$A \rightarrow c \bullet; d \perp$	应用 $A \rightarrow c$
q_3	$S \rightarrow a A \bullet d; \perp$	$\delta(q_3, d) = q_4$
q_4	$S \rightarrow a A d \bullet; \perp$	应用 $S \rightarrow a A d$
q_5	$S \rightarrow b \bullet B d; \perp$	$\delta(q_5, B) = q_7$
	$B \rightarrow \bullet c; d \perp$	$\delta(q_5, c) = q_6$
q_6	$B \rightarrow c \bullet; d \perp$	应用 $B \rightarrow c$
q_7	$S \rightarrow b B \bullet d; \perp$	$\delta(q_7, d) = q_8$
q_8	$S \rightarrow b B d \bullet; \perp$	应用 $S \rightarrow b B d$

7.2.3 归约与移进操作

FSA 的停机状态是那些包含一个其分隔符位于最右端的项目的状态，这一项目就是 LR 分析程序必须应用的产生式。栈的顶部恰好与待应用的产生式右部匹配，它将被该产生式左部的非终结符取代，这一动作称为“应用”或“归约”，因为产生式右部（又称句柄）被归约为非终结符。然后，FSA 又从栈底的状态 q_0 重新开始工作。

我们可用刚才构造的表格分析文法 G_{41} 的语言中的一个串 bcd 。分析工作从状态 0（即 q_0 ）开始且栈为空，FSA 所处的（空栈）栈顶不是一个停机状态，因而读入 b 并压入栈中。接着还是从状态 0 出发，FSA 选择沿 b 到达状态 5 的变迁，然后到达栈顶；此时要求读入另一单词，

这次将 c 压入栈中, 现在栈中有两个单词: bc 。然后再次从栈底的状态 0 重新启动 FSA, 沿变迁 b 快速前进到状态 5, 再沿变迁 c 到达状态 6; 状态 6 是一个停机状态, 与之关联的是产生式 $B \rightarrow c$, 因而从栈中弹出 c 并以非终结符 B 取而代之, 形成的新格局是栈中有 bB 。这时, 由栈底向上的状态序列为 0-5-7; 状态 7 是一个读入状态, 即它不归约任何产生式, 而是若 FSA 到达栈顶时处于该状态, 则读入下一单词并压入栈中。在栈中内容为 bBd 时 FSA 再次重新启动, 得到状态序列 0-5-7-8; 由于状态 8 是一个停机状态, 与之关联的产生式为 $S \rightarrow bBd$, 因而栈顶的三个符号被弹出, 并被非终结符 S 取代。 S 是目标符号, 因此分析程序接受该输入串。

当 FSA 到达栈顶时不是处于一个停机状态, 则读入下一输入符号并压入栈中。该过程又称“移进”, 因为输入单词从输入串的左边移到栈中。

7.3 冲突

如果 FSA 到达栈顶时所处的状态包含从两个不同产生式得到的归约项目, 或包含一个归约项目加上至少一个移进项目 (其中分隔符不是位于项目串的右端), 那么会出现“归约—归约”冲突或“移进—归约”冲突 (有时也称“读入—应用”冲突, 以与另一套术语保持一致)。在表 7-4 所示的文法 G_{41} 例子中, 不存在这样的冲突。一个 $LR(k)$ 分析程序中的任何“归约—归约”冲突均可基于在构造过程中携带的 *Follow* 集得到解析。这些 *Follow* 集在构造时就是接在每一产生式后面的 k 个符号; 由于文法是 $LR(k)$ 的, 冲突肯定可被解析。如果包括移进项目中分隔符右边的串的 $First_k$, 则可类似地解析“移进—归约”冲突。因而, 对于一个项目:

$$A \rightarrow u \bullet v; x$$

选择移进的向前看符号集将包括 $First_k(vx)$ 。如果分析程序的构造过程无法针对“归约—归约”冲突或“移进—归约”冲突生成互不相交的、 k 个符号的向前看符号集, 那么这一分析程序就不是 $LR(k)$ 的; 如果不存在这类冲突, 那么文法就是 $LR(0)$ 的。

7.4 例子: 文法 G_2 的冲突解析

再看另一个演示了如何解析一些冲突的例子, 试为文法 G_2 构造一个 $LR(1)$ 分析程序。我们使用一个特殊的目标符号 G , 以避免出现识别一个文法的接受状态时目标符号出现在某一产生式右部的问题。因而, 起始状态 q_0 的核包括一个项目:

$$G \rightarrow \bullet E; \perp$$

闭包运算将添加 E 、 T 和 F 的所有产生式, 以及因左递归的非终结符 E 和 T 的 *Follow* 集而产生的另外一些项目。为使表达方式更加紧凑, 仅在其 *Follow* 集有区别的项目被集中在同一行:

$$E \rightarrow \bullet E + T; \perp, +$$

$$E \rightarrow \bullet T; \perp, +$$

$$T \rightarrow \bullet T * F; \perp, +, *$$

$$T \rightarrow \bullet F; \perp, +, *$$

$$F \rightarrow \bullet (E); \perp, +, *$$

$$F \rightarrow \bullet n; \perp, +, *$$

第一条变迁沿非终结符 E 转入状态 q_1 , 这将前两个项目 (分隔符已前移) 作为核:

$G \rightarrow E \bullet ; \perp$
 $E \rightarrow E \bullet + T ; \perp, +$

此时，第一个项目是一个归约项目，而第二个是一个移进项目。然而，其中的“移进—归约”冲突很容易解析，因为归约项目的向前看符号集是文件结束符，而移进项目的向前看符号集则是跟在分隔符之后的单词“+”。

留意这一分隔符代表了 FSA 扫描输入串时的抽象“读磁头”，因而对冲突的解析是基于跟随在分隔符之后的串。仅当分隔符至串尾少于 k 个符号的情况下，我们才携带分号后的向前看符号集。在本例中，归约项目的右端有一个分隔符，因而它将依赖于向前看符号集，而移进项目的分隔符右边还有另一“+”须经过（忽略了向前看符号集中距离较远的文件结束符），因此两个集合是不相交的。读者自己可验证一下，表 7-5 是一个完整的构造过程，并且其余四个“移进—归约”冲突的解析也是同样地基于互不相交的向前看符号集。

表 7-5 文法 G_2 的 LR(1)分析程序状态

状 态	项 目	变 迁	向前看符号
q_0	$G \rightarrow E \bullet ; \perp$	$\delta(q_0, E) = q_1$	
	$E \rightarrow E \bullet + T ; \perp, +$		
	$E \rightarrow \bullet T ; \perp, +$	$\delta(q_0, T) = q_2$	
	$T \rightarrow \bullet T * F ; \perp, +, *$		
	$T \rightarrow \bullet F ; \perp, +, *$	$\delta(q_0, F) = q_3$	
	$F \rightarrow \bullet (E) ; \perp, +, *$	$\delta(q_0, () = q_4$	
	$F \rightarrow \bullet n ; \perp, +, *$	$\delta(q_0, n) = q_5$	
q_1	$G \rightarrow E \bullet ; \perp$	归约 $G \rightarrow E$	{ \perp }
	$E \rightarrow E \bullet + T ; \perp, +$	$\delta(q_1, +) = q_6$	{ $+$ }
q_2	$E \rightarrow T \bullet ; \perp, +$	归约 $E \rightarrow T$	{ $\perp, +$ }
	$T \rightarrow T \bullet * F ; \perp, +, *$	$\delta(q_2, *) = q_7$	{ $*$ }
q_3	$T \rightarrow F \bullet ; \perp, +, *$	归约 $T \rightarrow F$	
q_4	$F \rightarrow (\bullet E) ; \perp, +, *$	$\delta(q_4, E) = q_8$	
	$E \rightarrow \bullet E + T ;), +$		
	$E \rightarrow \bullet T ;), +$	$\delta(q_4, T) = q_9$	
	$T \rightarrow \bullet T * F ;), +, *$		
	$T \rightarrow \bullet F ;), +, *$	$\delta(q_4, F) = q_{10}$	
	$F \rightarrow \bullet (E) ;), +, *$	$\delta(q_4, () = q_{11}$	
	$F \rightarrow \bullet n ;), +, *$	$\delta(q_4, n) = q_{12}$	
q_5	$F \rightarrow n \bullet ; \perp, +, *$	归约 $F \rightarrow n$	
q_6	$E \rightarrow E + \bullet T ; \perp, +$	$\delta(q_6, T) = q_{13}$	
	$T \rightarrow \bullet T * F ; \perp, +, *$		
	$T \rightarrow \bullet F ; \perp, +, *$	$\delta(q_6, F) = q_3$	
	$F \rightarrow \bullet (E) ; \perp, +, *$	$\delta(q_6, () = q_4$	
	$F \rightarrow \bullet n ; \perp, +, *$	$\delta(q_6, n) = q_5$	
q_7	$T \rightarrow T * \bullet F ; \perp, +, *$	$\delta(q_7, F) = q_{14}$	
	$F \rightarrow \bullet (E) ; \perp, +, *$	$\delta(q_7, () = q_4$	
	$F \rightarrow \bullet n ; \perp, +, *$	$\delta(q_7, n) = q_5$	
q_8	$F \rightarrow (E \bullet) ; \perp, +, *$	$\delta(q_8,)) = q_{15}$	
	$E \rightarrow E \bullet + T ;), +$	$\delta(q_8, +) = q_{20}$	
q_9	$E \rightarrow T \bullet ;), +$	归约 $E \rightarrow T$	{ $), +$ }
	$T \rightarrow T \bullet * F ;), +, *$	$\delta(q_9, *) = q_{16}$	{ $*$ }

(续)

状 态	项 目	变 迁	向前看符号
q ₁₀	T → F • ;), +, *	归约 T → F	
q ₁₁	F → (• E ;), +, *	δ (q ₁₁ , E) = q ₁₇	
	E → • E + T ;), +		
	E → • T ;), +	δ (q ₁₁ , T) = q ₉	
	T → • T * F ;), +, *		
	T → • F ;), +, *	δ (q ₁₁ , F) = q ₁₀	
	F → • (E) ;), +, *	δ (q ₁₁ , () = q ₁₁	
	F → • n ;), +, *	δ (q ₁₁ , n) = q ₁₂	
q ₁₂	F → n • ;), +, *	归约 F → n	
q ₁₃	E → E + T • ;), +	归约 E → E + T	{), + }
	T → T • * F ;), +, *	δ (q ₁₃ , *) = q ₇	{ * }
q ₁₄	T → T * F • ;), +, *	归约 T → T * F	
q ₁₅	F → (E) • ;), +, *	归约 F → (E)	
q ₁₆	T → T * • F ;), +, *	δ (q ₁₆ , F) = q ₁₈	
	F → • (E) ;), +, *	δ (q ₁₆ , () = q ₁₁	
	F → • n ;), +, *	δ (q ₁₆ , n) = q ₁₂	
q ₁₇	F → (E •) ;), +, *	δ (q ₁₇ ,)) = q ₁₉	
	E → E • + T ;), +	δ (q ₁₇ , +) = q ₂₀	
q ₁₈	T → T * F • ;), +, *	归约 T → T * F	
q ₁₉	F → (E) • ;), +, *	归约 F → (E)	
q ₂₀	E → E + • T ;), +	δ (q ₂₀ , T) = q ₂₁	
	T → • T * F ;), +, *		
	T → • F ;), +, *	δ (q ₂₀ , F) = q ₁₀	
	F → • (E) ;), +, *	δ (q ₂₀ , () = q ₁₁	
	F → • n ;), +, *	δ (q ₂₀ , n) = q ₁₂	
q ₂₁	E → E + T • ;), +	归约 E → E + T	{), + }
	T → T • * F ;), +, *	δ (q ₂₁ , *) = q ₁₆	{ * }

7.5 在栈中保存状态

在 PDA 运行的每一步都让状态机遍历栈中的每一个符号是相当低效的，实际上第 2 章下推自动机的定义是拒绝此类浪费的。然而易见，LR(k) 的 PDA 的每一次移动最多只将一个符号压入栈中（尽管可能弹出多个符号）；因而除最后一个符号的压栈之外，对 PDA 的每一次移动，FSA 都将在整个栈中退回其步骤。如果扩展 PDA 栈的字母表以包括 FSA 的所有状态，则可在栈中每一符号之后插入移进该符号后的 FSA 状态，如图 7-3 所示。PDA 每一次移进或归约将一个新符号压入栈中之后，FSA 可从这一新符号之下（左边）的状态重新启动，从这里出发仅运行一步，然后将新状态压入在这一新的栈符号之上。PDA 根据 FSA 压入的状态的指示，立即准备好移进或归约操作，并且这一过程不断重复。如果现在接纳 FSA 的状态作为 PDA 的状态，则新的 PDA 符合本书的定义，惟一例外是允许一次从栈中弹出多个符号。

追踪图 7-3 对文法 G₂ 的输入串 $n + n * n$ 的分析，可发现栈顶（右端）总是包含 PDA 的当前状态。第 0 行的状态是 q₀，这是一个移进状态，因而从输入串移进一个符号（单词 n ）到栈中。表 7-5 中状态 0 的项目列表表明读入 n 后应前进到状态 5，使得 PDA 进入第 1 行。状态 5 是一个归约状态，因而产生式右部的所有符号（即单个 n ）从栈中弹出，并代之以相应的非终

结符 F；栈中的上一状态恰好是状态 0，因而在状态 q_0 的项目列表中可找到沿 F 出发的变迁，使得 PDA 进入第 2 行的状态 3。重复一次该过程，但在状态 q_2 （第 3 行）时状态表要求查看向前看符号；下一输入符号是“+”，故选择 PDA 执行归约动作。在第 8 行，分析程序再次要求向前看，选择结果是执行移进动作。

分析程序的栈				未读的下一输入符号		分析程序和动作
0				0	$n + n * n$	移进
1			0	$n, 5$	$+ n * n$	归约
2			0	F, 3	$+ n * n$	归约
3			0	T, 2	$+ n * n$	归约（向前看符号为“+”）
4			0	E, 1	$+ n * n$	移进
5		0	E, 1	$+, 6$	$n * n$	移进
6	0	E, 1	$+, 6$	$n, 5$	$* n$	归约
7	0	E, 1	$+, 6$	F, 3	$* n$	归约
8	0	E, 1	$+, 6$	T, 13	$* n$	移进（向前看符号为“*”）
9	0	E, 1	$+, 6$	T, 13	$*, 7$	移进
10	0	E, 1	$+, 6$	T, 13	$*, 7$	归约
11	0	E, 1	$+, 6$	T, 13	$*, 7$	归约
12		0	E, 1	$+, 6$	T, 13	归约（向前看符号为“ \perp ”）
13			0	E, 1		归约（向前看符号为“ \perp ”）
14			0	G, 1		接受

图 7-3 将 FSA 状态置于 PDA 栈中后，使用文法 G_2 分析串 $n + n * n$

无论分析程序何时归约一条产生式，它都无需校验栈中的符号是否与产生式右部的符号匹配，因为 FSA 已经完成了这些检查工作；只要计算出正确的栈中符号数目，并直接删除它们就足够了。归约动作删除符号后的栈顶状态编号变成分析程序的新状态，直至新压入的非终结符执行一次“移进”动作，亦即从栈中最后露出的状态编号的项目列表中查找下一状态。

7.6 其他 LR(k)分析程序：SLR

尽管 LR(k)分析程序的构造过程可保证构建一个确定的分析程序（假如存在这样的分析程序），但它往往导致状态表非常庞大。为削减状态表的大小，LR(k)语言设置了若干限制。这些是真正的限制，因为一种 LR(k)算法可为某一文法构造一个确定的分析程序，但在添加了这些限制的情况下却有可能无法构造。然而，通常可围绕这些限制来设计文法，正如本书第 4 章对 LL(k)文法的处理，并且这些限制通常不如 LL(k)的那么严格。LR 分析表构造的所有变形均使用相同的 PDA 状态序列发生器，它们之间的惟一区别在于分析表是如何构造的；这反过来又影响了分析表的大小。

1965 年，DeRemer 设计了一种“简单 LR”（即 SLR）分析程序。SLR 分析表的构造方式本质上与 LR 分析表相同，只是在构造过程中不携带 Follow 集信息。在出现“归约—归约”或“移进—归约”冲突时，改为直接利用相关非终结符的 Follow 集（由第 4 章的 LL(k)算法计算）。

这一做法有两个效果。首先，项目列表不再根据其 Follow 集区分开来，因此在文法 G_2 的构造过程中状态 q_2 和 q_9 将无法区分，从而也不会有各自独立的状态；类似地，状态 q_4 和 q_{11} 也是无法区分的。这种不可区分性会在状态对之间传播，例如 $q_8 - q_{17}$ 、 $q_{15} - q_{19}$ 、 $q_6 - q_{20}$ 、 $q_{13} - q_{21}$ 、 $q_3 - q_{10}$ 、 $q_5 - q_{12}$ 、 $q_7 - q_{16}$ 、 $q_{14} - q_{18}$ 等。因而 SLR 分析表（如表 7-6 所示）仅有 12 个状态，

而不是 22 个状态。对于更复杂的语言而言，状态数量的下降会更为可观。

表 7-6 文法 G_2 的 SLR(1) 分析程序状态

状 态	项 目	变 迁	向前看符号
q_0	$G \rightarrow \bullet E$ $E \rightarrow \bullet E + T$ $E \rightarrow \bullet T$ $T \rightarrow \bullet T * F$ $T \rightarrow \bullet F$ $F \rightarrow \bullet (E)$ $F \rightarrow \bullet n$	$\delta(q_0, E) = q_1$ $\delta(q_0, T) = q_2$ $\delta(q_0, F) = q_3$ $\delta(q_0, () = q_4$ $\delta(q_0, n) = q_5$	
q_1	$G \rightarrow E \bullet$ $E \rightarrow E \bullet + T$	归约 $G \rightarrow E$ $\delta(q_1, +) = q_6$	$\{\perp\}$ $\{+\}$
q_2	$E \rightarrow T \bullet$ $T \rightarrow T \bullet * F$	归约 $E \rightarrow T$ $\delta(q_2, *) = q_7$	$\{), +, \perp\}$ $\{*\}$
q_3	$T \rightarrow F \bullet$	归约 $T \rightarrow F$	
q_4	$F \rightarrow (\bullet E)$ $E \rightarrow \bullet E + T$ $E \rightarrow \bullet T$ $T \rightarrow \bullet T * F$ $T \rightarrow \bullet F$ $F \rightarrow \bullet (E)$ $F \rightarrow \bullet n$	$\delta(q_4, E) = q_8$ $\delta(q_4, T) = q_2$ $\delta(q_4, F) = q_3$ $\delta(q_4, () = q_4$ $\delta(q_4, n) = q_5$	
q_5	$F \rightarrow n \bullet$	归约 $F \rightarrow n$	
q_6	$E \rightarrow E + \bullet T$ $T \rightarrow \bullet T * F$ $T \rightarrow \bullet F$ $F \rightarrow \bullet (E)$ $F \rightarrow \bullet n$	$\delta(q_6, T) = q_{13}$ $\delta(q_6, F) = q_3$ $\delta(q_6, () = q_4$ $\delta(q_6, n) = q_5$	
q_7	$T \rightarrow T * \bullet F$ $F \rightarrow \bullet (E)$ $F \rightarrow \bullet n$	$\delta(q_7, F) = q_{14}$ $\delta(q_7, () = q_4$ $\delta(q_7, n) = q_5$	
q_8	$F \rightarrow (E \bullet)$ $E \rightarrow E \bullet + T$	$\delta(q_8,) = q_{15}$ $\delta(q_8, +) = q_6$	
q_{13}	$E \rightarrow E + T \bullet$ $T \rightarrow T \bullet * F$	归约 $E \rightarrow E + T$ $\delta(q_{13}, *) = q_7$	$\{), +, \perp\}$ $\{*\}$
q_{14}	$T \rightarrow T * F \bullet$	归约 $T \rightarrow T * F$	
q_{15}	$F \rightarrow (E) \bullet$	归约 $F \rightarrow (E)$	

现在只有三个冲突状态： q_1 、 q_2 和 q_{13} ；这三个冲突均为“移进—归约”冲突，其中待移进的是一个终结符。一个文法是 SLR(k) 的，如果对每一个含如下项目的“移进—归约”冲突状态：

$$A \rightarrow x \bullet$$

$$B \rightarrow y \bullet z$$

(其中 A 不一定是与 B 不同的非终结符)，均满足 $Follow_k(A)$ 与 $First_k(First_k(z) Follow_k(B))$ 无公共的串；并且对每一个含如下项目的“归约—归约”冲突状态：

$$A \rightarrow x \bullet$$

$$B \rightarrow y \bullet$$

均满足 $Follow_k(A)$ 与 $Follow_k(B)$ 无公共的串。由于 $Follow_1(G)$ 不包含 “+” 且 $Follow_1(E)$ 不包含 “*”，所以文法 G_2 是 SLR(1) 的。

有许多真实程序设计语言的无二义文法是 LR(1) 的，但不是 SLR(1) 的；虽然 Modula-2 语言不属于这类情况，但 C 语言却是这样的语言。与其用一种真实程序设计语言的复杂文法演示这一问题，不如用一个虚构的简单例子 G_{42} 来说明：

$$\begin{array}{ll} S \rightarrow a A a & A \rightarrow c \\ S \rightarrow b A b & B \rightarrow c b \\ S \rightarrow a B b & \end{array}$$

当根据 SLR 算法构造其分析表时，会遇到一个“归约—归约”冲突状态，其中包含以下两个项目：

$$\begin{array}{l} A \rightarrow c \bullet \\ B \rightarrow c \bullet b \end{array}$$

从整个文法可得 $Follow_1(A) = \{a, b\}$ ，因而通过考察非终结符 A 的 Follow 集无法解析冲突。但如果使用 LR 算法，则在构造过程中 A 所携带的局部 Follow 集此时仅包含 a，因而“归约—归约”冲突可通过单个向前看符号得到解析。

7.7 LALR(k)分析程序

LR 分析表的大小远大于 SLR 分析表，所以具有中间能力的分析表构造算法更合乎需要。1969 年，Korenjak 提出从 LR(k) 分析表出发，可以将具有相同核的所有状态合并为单个状态。由于在各种情况下每一项目的第一部分（即在产生式串中插入分隔符后所得的那一部分）均相同，所合并的只是状态编号和向前看符号集。合并后的状态中的向前看符号集是所有被合并状态的向前看符号集的并集。如果在每一种情况下合并产生的状态均不存在不可解析的冲突，则称该文法为 LALR(k) 的，通常读作“laller”（韵同“valor”）；尽管在逻辑上它应命名为“Look-Ahead SLR”或“Merged LR”，但这一首字母缩写词却自相矛盾地代表了“Look-Ahead LR”。LALR 状态表的大小与 SLR 的相同，但由于在构造过程中保留了向前看符号集，它能够解析一些 SLR 无法解析的冲突。

作为一个例子，再次考虑表 7-5 中文法 G_2 的 LR(1) 状态表。状态 q_2 与 q_9 合并后，产生的状态 q_{29} 具有相同的项目，但向前看符号集扩展为同时包含串结束符和右括号：

$$\begin{array}{lll} q_{29} & E \rightarrow T \bullet ; \perp,), + & \text{归约 } E \rightarrow T \quad \{ \perp,), + \} \\ & T \rightarrow T \bullet * F ; \perp,), +, * & \delta(q_{29}, *) = q_{716} \quad \{ * \} \end{array}$$

注意，由于状态 q_7 和 q_{16} 也被合并，故符号 “*” 的移进变迁转入合并后的状态 q_{716} 。

如果将 LALR(1) 构造过程用于一个 SLR(1) 文法，则生成的分析程序与由 SLR 构造过程产生的分析程序是同构的。但是，它比 SLR 更强大，因为有一些 LR(k) 文法不是 SLR(k) 的，却是 LALR(k) 的；例如，文法 G_{42} 不是 SLR(1) 的，但它却是 LALR(1) 的。然而，以下文法 G_{43} 是 LR(1) 的，但既不是 SLR(1) 的、也不是 LALR(1) 的：

$$\begin{array}{ll} S \rightarrow a A a & S \rightarrow a B b \\ S \rightarrow b A b & S \rightarrow b B a \\ A \rightarrow c & B \rightarrow c \end{array}$$

通过 SLR 算法构造其分析表时,会遇到一个“归约—归约”冲突状态,其中包含以下两个项目:

$$A \rightarrow c \bullet$$

$$B \rightarrow c \bullet$$

从整个文法看, $Follow_1(A) = Follow_1(B) = \{a, b\}$, 因而通过考察 $Follow$ 集无法解析该冲突。但如果使用 LR 算法, A 和 B 的产生式会构造出不同的局部 $Follow$ 集, 因而可利用单个向前看单词解析这一“归约—归约”冲突。然而, 基于状态的核项目合并了状态之后, 又再次回到不可解析的“归约—归约”冲突。

大多数现代的分析程序生成工具使用 LALR(1)分析表构造算法。还有一些方法比构建整张 LR(1)状态表后再合并的做法更高效。特别是 LR 算法每次创建一个新状态时, LALR 算法可检查是否有可能合并, 如果是则立即合并; 这意味着有些向前看符号集需要额外的工作, 以传播那些经合并步骤添加到合并后状态中的向前看符号。

以文法 G_2 中状态 q_2 和 q_9 的合并为例, 在即将创建状态 q_9 时会出现合并状态。但从 q_2 出发, 沿“*”变迁到达的目标状态 q_7 应早已存在; LALR 构造算法没有理由再创建一个 q_{16} 作为新合并状态的基础。因而, 添加到合并后状态 q_2 的向前看符号集中的右括号必须同时传播给状态 q_7 , 然后传播给状态 q_5 和 q_4 , 接着传播到状态 q_8 、 q_6 和 q_3 。

7.8 自底向上分析程序的实现

目前尚未出现与递归下降方式等价的自底向上分析程序的手工实现。这类分析程序通常是一个表格驱动的状态机, 十分类似第 3 章中实现一个扫描程序时构造的 FSA。主要的实现决策是如何高效地对状态表进行编码; 当然, 还存在另外一些方法可提升性能。

常见的 LR 分析表是一个二维数组, 一维记录状态编号, 另一维记录栈字母表中的符号 (即所有终结符与非终结符的集合 $\Sigma \cup N$)。表中每一入口记录以下四种动作之一:

- (1) 移进: 转入状态 s 。
- (2) 归约: 弹出 n 个符号; 再将非终结符 P 压入栈中。
- (3) 接受。
- (4) 错误代码 e 。

移进动作从输入串中读入一个单词 (即取出向前看符号后, 调用扫描程序并以返回结果代替这一向前看符号), 并将该单词与状态 s 一起压入栈中。归约动作实际上有两步, 因为从栈中弹出 n 个符号后, 栈顶状态被指派“读入”一个非终结符 P ; 因而将 P 与一个新状态压入栈中, 就好像读入 P 一样, 尽管输入串并未受到影响。接受动作终止分析过程。

表中所有其他入口均填入报错动作, 报告输入串不属于该语言。对于编译程序设计人员而言, 一种好的做法是尽力产生清晰的出错信息, 从而用户不会在出现各种可能的编码错误时, 都看到一条通用且没有什么信息含量的“出现语法错误”之类的消息。

自底向上的分析程序通常由分析程序生成工具自动地构造, 因而负责解释分析表的状态机往往是分析程序框架的一部分, 基本上无需进行大的修改就复制到每一个生成的分析程序中。代码清单 7.1 给出了一个典型分析程序状态机的 Modula-2 代码。

代码清单 7.1 LR 分析程序的分析解释程序

```

FROM Somewhere IMPORT
    MaxStateNo, MaxStack, NTToken,                (* 常量、类型和变量 *)
    Nextoken, Getoken, Error, GetTheTable;         (* 过程 *)

TYPE
    ActionCode = (ReadStep, ApplyStep, Accept, ErrorOff);

VAR
    StackTop: INTEGER;
    Done: BOOLEAN;
    theTable: ARRAY [0 .. MaxStateNo], NTToken OF RECORD
        theAction: ActionCode;
        Semantics: INTEGER;
        NewNT: NTToken;
    END;
    theStack: ARRAY [0 .. MaxStack] OF RECORD
        Symbol: NTToken;
        State: INTEGER
    END;

PROCEDURE Parser;
BEGIN
    StackTop := 0;                                (* 初始化栈 *)
    theStack[StackTop].State := 0;
    Done := FALSE;
    GetTheTable(theTable);                        (* 可能以块方式从磁盘中读入 *)
    REPEAT
        WITH theTable[theStack[StackTop].State][Nextoken] DO
            CASE theAction OF
                ReadStep:
                    INC(StackTop);                  (* 压入栈中... *)
                    WITH theStack[StackTop] DO
                        Symbol := Nextoken;          (* 压入新单词 *)
                        State := Semantics;          (* 以及新状态 *)
                        Getoken                      (* 读入下一向前看符号 *)
                    END (* WITH *) |
                ApplyStep:
                    StackTop := StackTop - Semantics + 1; (* 弹出产生式右部 *)
                    WITH theStack[StackTop] DO
                        Symbol := NewNT;              (* 压入非终结符与状态 *)
                        State := theTable[theStack[StackTop - 1].State][NewNT].Semantics
                    END (* WITH *) |
                Accept:
                    Done := TRUE |
                ErrorOff:
                    Error(Semantics);                (* 发现既无移进、也无归约动作 *)
                    Done := TRUE
            END (* CASE *)
        END (* WITH *)
    UNTIL Done
END Parser;

```

7.9 出错恢复

迄今为止, 我们的分析程序在输入源代码中遇到一个语法错误时, 响应方式均为“玉石俱焚”, 意即放弃编译、报告错误、然后退出。以前, 这一响应方式受到某些典型用户的指责, 这些用户每天只会在批处理的大型计算机上尝试一次或两次编译。如果一个编译程序能够在发现错误后尝试恰当的恢复, 并尽可能地继续分析输入源程序, 则可使程序员有机会在重新提交程序再次编译之前修改多个错误。为此, 研究人员已在最常见的编码错误分析方面开展了大量工作, 这种分析不仅可帮助编译程序设计人员提高从语法错误中恢复的概率, 而且有利于产生更有信息含量的报错信息, 从而用户更容易作出相应的改正。

大多数常见的语法错误是由以下三种形式的单个单词(或单个字符)错误造成的: 丢失的单词、多余的单词以及被替代的单词。一种出错恢复的启发式方法可能是尝试补充一个丢失的单词, 跳过一个多余的单词, 以及用一个更合适的单词取代未能正确分析的单词。补充丢失的单词时最应格外小心。如果连续地插入单词, 分析程序可能只会在错误状态中循环而不会再移进任何输入, 这显然不是一种正确的响应方式。通常如果插入单个单词后仍无法修正问题以致成功地分析输入串, 那么就必须放弃单词丢失的这一假设。

出错恢复的启发式方法的一种变形是丢弃多余的单词, 称之为“应急模式”, 因为该方法强制丢弃单词, 直至遇到一个可识别的产生式边界(通常是一个分号); 栈也必须刷新, 从而分析程序可恢复到 **StatementList** 产生式的归约状态。这种恢复策略的典型做法是在发现错误后, 先跳过源文件中某些(通常较少的)部分, 然后在真正出错位置之后的下一条语句恢复对输入串的分析。

最复杂的出错恢复技术是由 Pennello 等人提出的所谓“向前移动”技术 [Pennello&DeRemer, 1978]。文法中添加了错误产生式, 分析程序生成工具创建出错恢复状态, 在这些状态尝试移进、丢弃单词和归约产生式, 直至出现某些可以继续工作的东西。

随着个人交互式工作站以及即时响应时间的出现, 潮流不再是复杂的出错恢复技术, 而是以一种玉石俱焚的终止方式, 在程序编辑器中将用户定位到发现错误的单词上。超快的编译速度使得这一技术相当实用, 因为用户可要求再次编译并在很短时间内得到下一个错误报告, 时间上不亚于平常的在程序清单中查找一条嵌入的出错消息。

应注意, 规范 $LR(k)$ 分析表的构造会得到最快的语法错误检测。LALR (以及 SLR 的更大扩展) 分析表导致分析程序在检测出一个错误之前, 可能经过若干归约步后前进若干状态。然而在正常情况下, 这一语法错误在移进任何输入符号之前就会被发现; 对于 LALR 而言, 这总是成立的。在这三种构造方式中, 分析程序均可正确地检测有错的输入串, 惟一区别在于这些错误有多快被发现。

7.10 LR 分析程序中的属性求值

与自顶向下分析相比, 在自底向上分析中直观地观察属性值的流向是很困难的。如第 5 章所述, 在分析过程中仅可对综合属性和从左到右的继承属性进行求值。属性值可添加到 PDA 的栈字母表中, 或可定义另一个栈以保存这些属性值; 它们将在栈帧中分配空间, 与块结构语言中编译得到的局部变量管理代码十分类似。

属性求值通常必须发生在 LR 分析程序状态机的归约步。此时，综合属性可直接从正被归约的产生式右部的终结符和非终结符的综合属性计算得到。那些沿分析树向上传递的综合属性，可存储在一个为当前压入分析程序栈中的非终结符而新建的栈帧中。

继承属性的实现更加困难，因为一般不知道它们继承了哪一个非终结符。在自顶向下的分析中，继承属性在分析树中的父结点组装；但在自底向上的分析中，该结点尚未存在。在非终结符之间传递从左到右继承属性的惟一实用方法也许是构造一个数据包，其中的字段包含了所有非终结符的所有可能的继承属性，然后在构建分析栈时将该数据包沿分析栈传播。每一个归约步合成的新属性值均在该数据包中占有一个位置；归约步将一个新的值放入该数据包，然后才将该值的一个新副本沿栈向上传递给下一终结符或非终结符。那些不影响该数据包中任何属性的单词和非终结符，只需原封不动地传递一个副本。该数据包一般不会太大，因为仅有少数属性（典型情况是只有符号表）需按此方式传播。图 7-4 演示了数据包中仅含单个从左到右属性的处理过程。



图 7-4 自底向上分析中的属性求值

大多数表格驱动的分析程序将全部语义动作收集到一个大的 **CASE** 语句中，由存储在分析表中的编号作为不同情况的索引。当模块化设计要求语义动作有各自独立的过程时，**CASE** 语句的项目将是带适当参数的过程调用。定义编译程序语义的属性文法相对新奇，且在自底向上分析程序中正确处理任意属性信息流有困难，这些均限制了基于属性文法的编译程序生成工具的可用性。应用最广泛的分析程序生成工具是 YACC，它将所有语义动作降级为嵌入在文法中的 C 语言代码。

小结

最大一类可确定地分析的语言须使用自底向上的分析技术。与自顶向下分析（从目标符号开始，自顶向下构造分析树，直至到达分析树的边缘，从而推导出语言中一个句子）相比，自底向上分析从底部（边缘或句子）构建分析树，一路向上构建到分析树的根。

自底向上分析使用一个下推列表（栈），它将输入符号压入栈中（移进），直至栈顶出现一个句柄；句柄是一个子串，它与用于推导串的文法中的某一重写规则的右部匹配。一旦在自底向上分析程序所用的栈的顶部出现一个句柄，就立即归约该句柄，即用该重写规则左部的非终结符取而代之。只要未发现

错误, 自底向上分析过程将继续, 直至扫描了整个输入串, 并且栈中仅留有一个目标符号。

高德纳在 1965 年提出的 LR 分析方法是最通用的。一个 LR(k) 分析程序从左到右扫描其输入串, 并生成一个逆序的最右推导; 在进行分析的决策时, 它最多只需 k 个向前看符号。遗憾的是, 事实证明对于定义一种实用程序设计语言的文法而言, 在 LR 分析中产生的分析表实在太大了。DeRemer 在 1969 年提出的 SLR (简单 LR) 分析程序解决了这一问题, 他注意到有时可以合并 LR 分析表中的行。Korenjak 在 1969 年提出的 LALR (向前看 LR) 分析技术以另一方式解决了 LR 分析表过大的问题。本章详细介绍了 SLR 和 LALR 两种分析技术, 最后讨论了一个 LR 分析程序中的属性求值问题。

关键术语

apply step (应用步 (归约步)) 自底向上分析程序的一个操作步骤, 通过应用文法中的一条重写规则, 将栈顶的句柄归约为一个非终结符。

bottom-up parsing (自底向上分析) 在一个上下文无关文法的分析程序中, 按照逆序的最右规范推导构造分析树, 从语言中的一个串开始, 将它归约为目标符号。

handle (句柄) 自底向上分析程序栈顶的、由终结符与非终结符组成的串, 与文法中的重写规则 (产生式) 的右部匹配。

item (项目) 文法中一条重写规则的副本, 对该副本的改动是添加了一个分隔符, 表示自底向上分析程序构造中 FSA 的读磁头。

closure (闭包运算) 对项目列表中那些分隔符恰好在一个非终结符左边的项目, 将文法中该非终结符的所有产生式添加到项目列表中。如果 $A \rightarrow u \bullet B v$ 是项目列表中的一个项目, 则闭包运算将添加形如 $B \rightarrow \bullet w$ 的项目, 其中 $B \rightarrow w$ 是文法中的一条产生式。

complete item (完成项目) 项目中分隔符是最右端的符号。

kernel (核) 构造 LR(k) 分析程序的状态时, 将分隔符向前移动一个符号后、但不执行闭包运算就得到的项目。仅比较两个项目集的核, 即可确定它们是否相同。

LALR(k) (LALR(k) 文法) 即 Lookahead LR(k) 文法, 其中合并后状态的向前看符号集是所有被合并状态的向前看符号集的并。

LR(k) grammar (LR(k) 文法) 该文法的分析程序从左到右扫描输入串, 按最右规范推导的逆序遍历, 最多在输入流中向前看 k 个符号即可完成确定的分析。

LR(k) parser (LR(k) 分析程序) 根据一个 LR(k) 文法构造的分析程序。

LR parsing (LR 分析) 参阅 bottom-up parsing 条目。

merged state (合并后状态) LALR(k) 分析程序构造过程中的一个状态, 将两个仅向前看符号集不同的 LR(k) 状态组合其项目后得到。

panic mode (应急模式) 一种出错恢复的启发式方法, 从输入流中丢弃单词, 直至找到一个可识别的重写规则边界。

predictive parser (预测分析程序) 一种自顶向下的分析程序。

read step (读入步) 自底向上分析程序的一个操作步骤, 读入下一输入符号, 并将它移进 (压入) 栈顶。

reduce step (归约步) 一步应用产生式的动作。

shift step (移进步) 即读入步。

SLR(k) (SLR(k) 文法) 可由 SLR(k) 分析程序识别的语言的文法, 其构造忽略了 LR(k) 构造过程中的向前看符号集, 并使用 $Follow_k$ 集解析 “移进—归约” 冲突。

top-down parser (自顶向下分析程序) 一个上下文无关文法的分析程序, 按最左规范推导构造分析树; 从目标符号出发, 将它扩展为语言中的串, 即输入串。又称预测分析程序。

练习

- 设第 7.6 节给出的文法 G_{42} 定义了语言 L , 使用该文法给出:
 - 属于 L 的某个串的最右推导。
 - 一个项目的例子。
 - 使用 SLR 算法构造的分析程序状态, 并指出存在“移进—归约”冲突的所有状态。
 - 使用 LR 算法构造的分析程序状态。
 - 比较上述 (c) 小题和 (d) 小题的状态表。
- 请指出:
 - 练习 1 (c) 小题的状态表中的核。
 - 表 7-5 的核。
- 考虑文法 $(\{a, b\}, \{S, A\}, P, S)$, 其中 P 是以下产生式集合:

$$S \rightarrow aS \mid bAS \mid a$$

$$A \rightarrow abA \mid a \mid b$$
 判断该文法是否:
 - 存在某个 k , 使得它是 $LR(k)$ 的 (若回答“是”则请指出 k 的值), 或
 - 存在某个 k , 使得它是 $SLR(k)$ 的, 或
 - 存在某个 k , 使得它是 $LALR(k)$ 的, 或
 - 存在某个 k , 使得它是 $LL(k)$ 的。
- 说明对于以下产生式定义的文法, 存在某个 k , 使得该文法是 $LALR(k)$ 的; 并请指出 k 的值。

$$S \rightarrow Aa \mid dAb \mid cb \mid dca$$

$$A \rightarrow c$$
- 判断练习 4 的文法是否对于同样的值 k 也是 $SLR(k)$ 的; 如果不是, 则指出其中至少有一个“移进—归约”冲突状态, 该冲突无法用 $Follow_k$ 解析。
- 找出一个是 $LALR(k)$ 的、但不是 $LL(k)$ 的文法例子, 从而说明并非所有 $LALR(k)$ 文法都是 $LL(k)$ 的。
- 说明并非所有 $LL(k)$ 文法都是 $LALR(k)$ 的。
- 说明并非所有 $LR(0)$ 文法都是 $SLR(0)$ 的。
- 说明任一 $LL(k)$ 文法也是一个 $LR(k)$ 文法。

复习小测验

指出下列陈述是否正确。

- 在 LR 分析中, 一个应用步与一个归约步是相同的。
- 在一个 LR 分析过程中, 如果下一分析符号被读入并移进, 那么被读入的输入符号将从栈顶弹出。
- 并非每一个 $LALR(k)$ 文法都是一个 $SLR(k)$ 文法。
- 并非所有 $SLR(k)$ 文法都是 $LALR(k)$ 文法。
- 一个 $LR(k)$ 文法是一个上下文无关文法。
- 每一个 $LL(k)$ 文法都是一个 $LR(k)$ 文法。
- 以下产生式的文法是 $LR(0)$ 的:

$$A \rightarrow Ax \quad A \rightarrow x$$
- 以下产生式的文法是 $LR(1)$ 的:

$$A \rightarrow AaAb \quad A \rightarrow \varepsilon$$
- 一个文法是 $LR(k)$ 的, 如果从右到左扫描输入串, 并在句柄右端之外前进最多不超过 k 个符号, 即可在最右推导中确定每一右句型的句柄, 并找出以哪一个非终结符取代句柄。

编译程序实验项目

1. 为 Itty Bitty Modula 语言的文法构造一个 SLR(1)分析表, 暂时不理睬语义动作。使用类似于代码清单 7.1 的状态机, 用一些以 Itty Bitty Modula 语言编写的小程序测试你的分析程序。请验证语法上不正确的程序会被拒绝。
2. 在第 5、6 章开发的属性文法基础上, 为你的自底向上分析程序添加语义。
3. 设计一个 LALR(k)分析程序生成工具。编写一个 LL(1)属性文法作为其输入, 并在 TAG 编译程序上编译该文法, 然后在你的分析程序生成工具上编译该文法。

进一步阅读

Aho, A.V. & Ullman, J.D. *The Theory of Parsing, Translation, and Compiling: Vol. 2, Compiling*. Englewood Cliffs, NJ: Prentice Hall, 1973.

参阅第 7.4.1 小节 (第 622~627 页) 对 SLR(k)文法和原理的介绍; 参阅第 7.4.2 小节 (第 627~645 页) 对 LALR(k)文法的介绍。

Bermudez, M.E. & Schimpf, K.M. "On the (Non-) Relationship Between SLR(1) and NQLALR(1) Grammars." *ACM Transactions on Programming Languages and Systems*, Vol.10, No.2 (April 1988), pp.338-342.

NQLALR 中的 NQ 表示“不那么”(Not Quite)。NQLALR 文法由 DeRemer 和 Pennello 在 1982 年提出, 它基于状态之间、而不是非终结符变迁之间的相似关系。Bermudez 和 Schimpf 指出 SLR(1)文法并非 NQLALR(1)文法的一个子集。

DeRemer, F.L. "Simple LR(k) Parsing." *Communications of the ACM*, Vol.14, No.7 (1969), pp.453-460.
介绍了 SLR 分析方法。

DeRemer, F.L. & Pennello, T. "Efficient Computation of LALR(1) Look-Ahead Sets." *ACM Transactions on Programming Languages and Systems*, Vol.4, No.4 (October 1982), pp.615-649.

介绍了 NQLALR(1)文法 (参阅 Bermudez 和 Schimpf 的文献), 它被认为包含了所有 SLR(1)文法。

Hopcroft, J.E. & Ullman, J.D. *Introduction to Automata Theory, Languages, and Computation*, Reading, MA: Addison-Wesley, 1979.

参阅第 10.6 小节关于 LR(0)文法的介绍, 特别是第 248~252 页。

Ives, F. "Unifying View of Recent LALR(1) Lookahead Set Algorithms." *ACM SIGPLAN Notices*, Vol.21, No.7 (July 1986), pp.131-135.

Knuth, D.E. "On the Translation of Languages from Left to Right." *Information and Control*, Vol.8, No.6 (1965), pp.607-639.

Korenjak, A.J. "A Practical Method for Constructing LR(k) Processors." *Communications of the ACM*, Vol.12, No.11 (1969), pp.612-623.

介绍了 LALR 分析方法。

Pennello, T.J. "Very Fast LR Parsing." *ACM SIGPLAN Notices*, Vol.21, No.7 (July 1986), pp.145-151, containing *Proceedings of the SIGPLAN 1986 Symposium on Compiler Construction*.

报告了在一台类似 VAX 11/780 计算机上每分钟处理 50 万行代码 (每分钟至少处理 4 万行) 的分析速度。这一性能提升得益于将分析程序的有穷状态控制代码翻译为汇编语言。

Pennello, T.J. & DeRemer, F.L. "A Forward Move Algorithm for LR Error Recovery." *Fifth Annual ACM Symposium on Principles of Programming Languages* (1978), pp.241-254.

第 8 章 变换属性文法

本章旨在：

- 介绍树变换文法
- 探讨使用树文法将抽象语法树变换为不同形状的树
- 区别翻译文法与变换文法
- 介绍变换属性文法 (TAG)，它既是一个树文法，也是一个属性文法
- 介绍一种确定属性求值次序的简化方法
- 指出如何将编译程序前端的串文法链接到定义优化与代码生成的树文法
- 探讨基于变换的代码优化
- 展示如何将属性文法用于数据流分析
- 综述实用的变换优化

8.1 简介

尽管第 1 章引入了抽象语法树 (AST) 作为被编译的输入源程序的语法结构概念表示，我们还应注意到编译程序通常并不真正地构建一个 AST。在本章，我们主要考虑那些希望在内存中构造出表示 AST 的数据结构的情况。

8.2 程序的树表示

一棵抽象语法树正如其名所示，它从源程序中抽取的只是语法结构，既没有保留标识符或关键字的拼写，也没有存放空格、换行符、注释等。因而在图 8-1 中，Modula-2 语言的 **IF-THEN-ELSE-END** 语句的抽象表示与 Pascal 语言的 **if-then-else** 语句完全相同（惟一例外是子树的形式略有不同）。该结点连接了三个程序片段作为其子树：第一棵子树表示待求值的布尔表达式；第二棵子树表示所求值的表达式为真时执行的语句或语句序列；第三棵子树表示表达式为假时执行的另一语句序列。当然，这两棵语句子树本身既可以是条件语句结点，也可以是赋值语句或其他语句。

AST 与分析树在两个重要方面有所区别。首先，如前所述，保留字、标识符的拼写以及常量不复存在，只剩下足以作为树结构中结点标签的信息。在图 8-1 的表示中，消除了保留字 **THEN**、**ELSE** 和 **END**，还有标点符号以及标识符 **a**、**b** 和 **anysubroutine** 的拼写。

其次且更重要的是，消除了不含任何语义动作的文法非终结符。图 8-1 所示例子中，并未引用非终结符 **Expression** 或 **Factor**，尽管在分析任何表达式时肯定会涉及它们。非终结符 **Term** 仅在标签为 “+” 的结点有所表示，而这只是因为它产生加法的语义动作。一棵 AST 仅仅抽象了保持程序语义的那部分语法结构。一个树遍历自动机可遍历 AST，并在访问每一结点时生成代码（典型的是后序遍历，但也有大量的例外），生成的代码与第 6 章介绍的递归下降属性文法所产生的代码相同。

```

IF a < b THEN
  a := b + 5
ELSE
  anysubroutine(4, b)
END

```

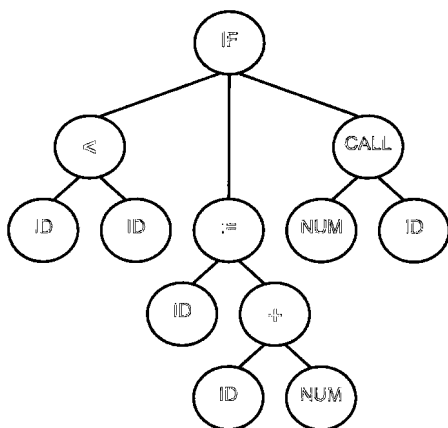


图 8-1 一个程序片段的树表示

如果我们的惟一目标是将分析过程与代码生成过程分离，那么 AST 不失为中间代码的一种合理表示；一旦将意义重大的代码优化映入眼帘，AST 就成为不可或缺的；这是因为许多优化工作的最简单执行方式就是对 AST 进行变换（整形）。

8.3 树变换文法

鉴于本书始终如一地强调文法在编译程序设计中的重要性，由文法作为定义树变换的最简单方式也就不足为奇，这类文法称为树变换文法（TTG）。TTG 是更大的树文法类集中的一个子集。

树文法与我们迄今所见文法的区别在于，其输入字母表由树结点及其连接组成，而不是 ASCII 字符或扫描一个线性串得到的单词。由于树是一种二维结构，因此树文法必须识别结点的结构，而不仅是单词的线性串，尽管书写文法的产生式时，典型的树表示方式是前缀波兰表示法。

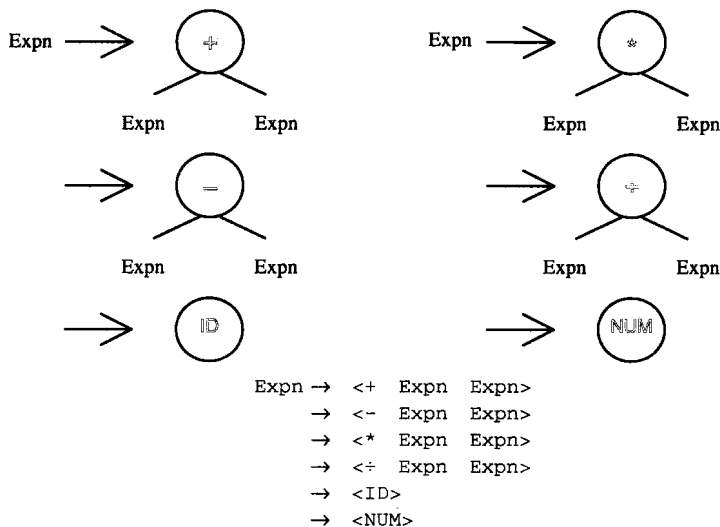


图 8-2 表达式树文法的片段，同时给出了其图形表示和文本表示

图 8-2 展示了一个识别表达式树的树文法片段。请注意，该文法无须利用圆括号或中间非终结符即可无二义地展示运算符层次，这是树结构的固有特性。根据这一文法，一个表达式要

么由一个标识符或数字的叶结点组成，要么由一个简单的运算符结点以及两棵表达式子树组成。图 8-3 展示了该文法生成的一个表达式。

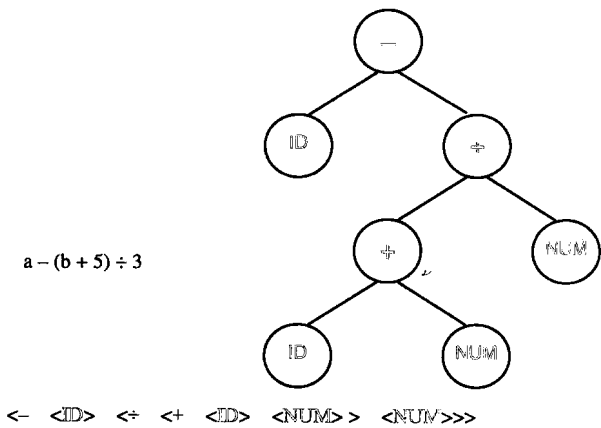


图 8-3 图 8-2 文法生成的表达式树

在编译程序构造中，只是识别（即分析）由一个树文法生成的抽象语法树并不会特别有用，我们还希望能将它们变换为不同形状的书。考虑图 8-3 中的表达式树，并假设编译程序发现标识符 **b** 被声明为常量 4，则可将 ID 结点变换为 NUM 结点以优化该树；然后在计算正确的新常量值后，用单个 NUM 叶结点取代位于两个 NUM 结点之上的任一运算符结点。图 8-4 演示了这些变换步骤。图 8-5 展示了一个用于常量折叠的简单 TTG；所谓常量折叠，意即在编译时对常量表达式进行求值。

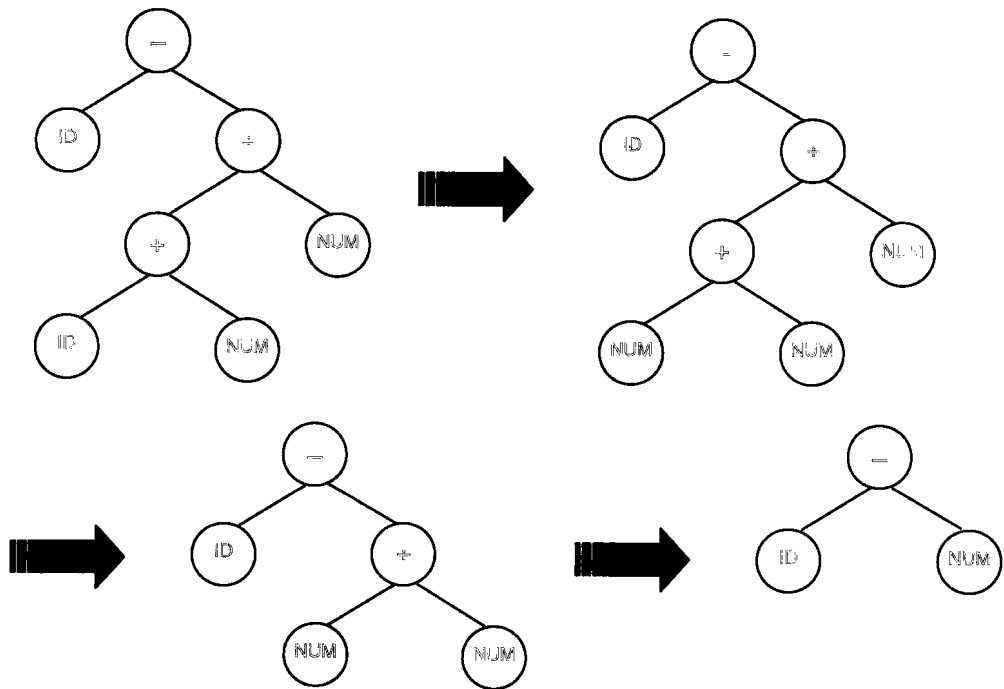


图 8-4 通过变换优化表达式树

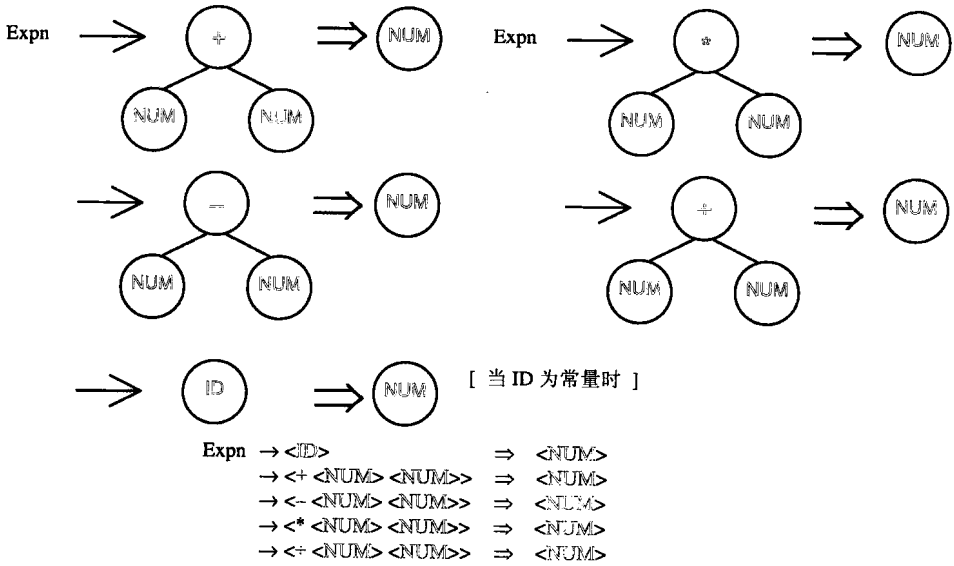


图 8-5 用于常量折叠的一个简单树变换文法

变换文法 T 是一个上下文无关文法，且其中每一产生式均有两个右部，这两个右部包含了相同的非终结符引用，尽管这些引用的次序不一定相同。每一产生式的第二右部由它左边的双箭头“ \Rightarrow ”指明，双箭头将它与第一右部分隔开。变换文法中的一条产生式抽象地看起来有些类似于如下的形式：

$$\text{nonterminal} \rightarrow \text{"first" rightpart} \Rightarrow \text{"second" rightpart}$$

如果删除所有的第二右部，则所剩下的就是一个普通的上下文无关文法，它所定义的语言称为“输入语言”；如果删除所有的第一右部，并将双箭头改为单箭头，则所剩下的仍是一个普通的上下文无关文法，它所定义的语言称为“输出语言”。当为识别输入语言而构造的分析程序应用文法中的某一产生式时，与该产生式右部匹配的子树将被输出语言中对应的同一产生式的右部取代，从而将输入语言变换为输出语言。

第一右部有时也称“匹配模板”，第二右部则称“生成器模板”。我们要求两个右部或模板的非终结符之间有一一对应关系，从而分析程序递归地遍历 AST 时，每一非终结符在输出语言中都占有一个位置。输入语言和输出语言中的终结符则无须有对应关系。

不难看出，代码生成程序属性文法的语义动作很容易写成变换文法的形式，其中第二右部编码了本书表示法中以方括号括住的终结符。实际上，如果我们没有非局部优化的需求，这已是一种自然且有效的表示法。然而我们并不打算这样做，而是直接去感受树变换属性文法的全部能力。

将属性文法用于定义一个编译程序如何通过对树表示的程序进行变换以实现优化，其原创性工作由慕尼黑技术大学的 Eickel、Ganzinger、Giegerich、Ripken 和 Wilhelm 等人在 1975 年开发编译程序生成工具 MUG2 时完成 [Madsen, 1975]。时至 1983 年，MUG2 已发展为一个基于属性文法的完整编译程序生成工具，它使用抽象语法树和带属性的树变换来处理属性求值问题。在 MUG2 中，通过由文法指定的变换，用附加在代码树上的信息标识一个属性；这些信息

实际上是程序树中的特殊叶结点,而不是高德纳所理解的属性[Knuth, 1968]。在美国, Burroughs 公司的 T. F. Payton 于 1982 年开发了一个语法和语义分析与生成系统,采用树变换文法将输入文法树映射为输出文法树; Payton 的系统将一个 TTG 翻译为一个执行翻译并生成相应求值程序的属性文法 [Payton, 1982]。

8.3.1 非生成的文法

尽管一个完整定义的变换文法在生成一个输入语言的同时,还生成一个可能截然不同的输出语言,但一个高效的编译程序在无须执行变换时(亦即第二右部与第一右部相同时),不应将大量时间浪费在树的遍历和变换中。因而我们区别“翻译文法”和“变换文法”这两个概念:翻译文法完整地生成输入语言和输出语言,而变换文法中输入语言和输出语言本质上是相同的语言。

翻译文法和变换文法均有二个右部的集合,但变换文法(TTG)中省略了将一棵树变换为自身的产生式。图 8-5 所示是一个 TTG,而不是一个翻译文法,因为它定义了如何变换两个常量之上的运算符结点,而不是指定如何变换一个或多个标识符之上的运算符结点,也未给出令一个标识符保持不变的情况。图 8-4 中的变换似乎是以一种投机的方式应用变换规则,只要有一个模板匹配即可;早期根据 TTG 构造的研究性编译程序也确实如此。我们将基于构造编译程序时所依据的变换属性文法,使用一种确定的算法查找树变换的机会。

变换属性文法(TAG)是一个树变换文法,进而它也是一个属性文法。这意味着文法中每一非终结符均定义了 0 个或多个属性。由于综合属性和继承属性均可能出现,属性求值次序对文法强加了一些限制:游离的非终结符不可沿树向上或向下传递任何属性。因而,我们要求 TAG 虽然不必是完全生成的,但它们必须生成一棵树,该树与完整的程序树具有相同的根结点,只是整棵子树消失了。TAG 无法生成一棵非连通的树,也无法生成从根结点不可达的任何树片段。因而,一个树变换属性文法是一个六元组 (Σ, N, P, S, A, E) , 其中:

Σ 由树的结点名(终结符)组成的字母表。

N 非终结符的集合。

P 产生式的集合。

S 目标符号,与树的根结点相关联。

A 属性的集合,满足每一属性恰好与一个非终结符相关联,并被指定为要么是一个综合属性,要么是一个继承属性。

E 断言的集合,这些断言限制和定义了每一产生式中属性的值。

TAG 中的每一产生式包括五部分:

$$n \rightarrow t_1 \quad s_1 \Rightarrow t_2 \quad s_2$$

其中:

n 是一个非终结符,集合 N 中的成员。

t_1, t_2 是树模板,在文本中写为 $i \text{ ":" } "<" \sigma t_k \text{ "*" } ">" \% e$, 其中

i 是一个(可选的)标识符,用于命名该子树。

σ 是结点名集合 Σ 的一个成员。

t_k 是在同一表示中的另一个子树模板。

e 是一个(可选的)树“装饰”,稍后将给出其定义。

文字符号“.”和“%”用于指明与它们关联的可选部分是否出现；文字符号“<”和“>”的作用类似于圆括号，作为树结点与其子树之间的分界。

s_1, s_2 是括在方括号中的语义动作序列，以及非终结符的引用（以它们所应用的子树名作为前缀）。

附录 B 给出了本书所用 TAG 的完整文法定义，目前我们主要关注它们的通用形式。

8.3.2 一个 TAG 例子

下面以一个小型 TAG 为例，考察如何基于常量表达式的剪枝完成对一个表达式树的变换。该例子将大致按图 8-5 组织，但新添加了属性断言并完善了产生式，从而使得该文法是完全生成的，如代码清单 8.1 所示。这里假设所有标识符均定义在一个继承的符号表中。

代码清单 8.1 一个简单的常量折叠 TAG

Expn ↓symbol ↑iscon ↑value

→<Plus left rite>

rite: Expn ↓symbol ↑iscon2 ↑value2

left: Expn ↓symbol ↑iscon1 ↑value1

[[iscon1 = true; iscon2 = false; iscon = false] left: Xform ↓value1
|[iscon2 = true; iscon1 = false; iscon = false] rite: Xform ↓value2
|[iscon1 = iscon2; iscon = iscon1; value = value1 + value2]]

→<Minus left rite>

rite: Expn ↓symbol ↑iscon2 ↑value2

left: Expn ↓symbol ↑iscon1 ↑value1

[[iscon1 = true; iscon2 = false; iscon = false] left: Xform ↓value1
|[iscon2 = true; iscon1 = false; iscon = false] rite: Xform ↓value2
|[iscon1 = iscon2; iscon = iscon1; value = value1 - value2]]

→<Star left rite>

rite: Expn ↓symbol ↑iscon2 ↑value2

left: Expn ↓symbol ↑iscon1 ↑value1

[[iscon1 = true; iscon2 = false; iscon = false] left: Xform ↓value1
|[iscon2 = true; iscon1 = false; iscon = false] rite: Xform ↓value2
|[iscon1 = iscon2; iscon = iscon1; value = value1 * value2]]

→<Divd left rite>

rite: Expn ↓symbol ↑iscon2 ↑value2

left: Expn ↓symbol ↑iscon1 ↑value1

[[iscon1 = true; iscon2 = false; iscon = false] left: Xform ↓value1
|[iscon2 = true; iscon1 = false; iscon = false] rite: Xform ↓value2
|[iscon1 = iscon2; iscon = iscon1; value = value1 / value2]]

→<ID>%name

[from ↓symbol ↓name ↑iscon ↑value]

→<NUM>%value

[iscon = true] ;

Xform ↓value

```

→ <Plus left rite> | <Minus left rite> | <Star left rite> | <Divd left rite>
⇒    <NUM>%value

→    <ID>%name
⇒    <NUM>%value

→    <NUM>%value ;           { 此处保持不变 }

```

文法首先遍历表达式树，以找出常量子树。我们采取的做法是向上携带一个综合属性表示一棵子树是否常量，而不是将大量时间浪费在对一个更大常量子树中的部分子树进行剪枝；当一个运算符连接一个常量与一个非常量时，通过将该子树发送给非终结符 **xform**，从而将该常量折叠成一个 **NUM** 结点。该非终结符不会遍历整棵子树，而是立即对该子树进行变换，用一个常量结点取而代之（除非它本身已是一个常量）。在该文法中，无需定义任何树的形状或名字；产生式中的树模板是匿名的，其子树已命名，但未展示其结构。

在代码清单 8.1 中，通过装饰区分了两类不同的结点。标识符结点用在符号表中找到的该标识符的名字装饰，这允许在有需要时可提取出它们的特性；常量结点用它们的数值装饰，一个常量子表达式的值可通过一个装饰常量结点的值以及一个从符号表中找到的常量标识符的值构造出来。对一棵子树进行变换时，新的常量结点将以一个适当的复合值装饰。

8.3.3 求值次序

代码清单 8.1 所示的常量折叠例子中，用一个非终结符 **Expn** 收集了足以对是否执行变换作出决策的信息，但由另一非终结符 **xform** 真正执行变换；这种分工是常见且合理的。注意，当 **xform** 表示为一个仍不是常量的结点时，该非终结符将无条件地执行变换；而仅当 **Expn** 有一个常量表达式需要变换时，该非终结符才会调用 **xform**。

在上述例子中，信息收集阶段的信息流次序基本上是自底向上的，未考虑从左到右的次序。在更多情况下，信息流本质上是从左到右、或从右到左的；本章稍后将讨论这两种类型的特定例子。当我们引入属性文法时，我们考虑几个需要从右到左或混合型信息流的文法。对于一个递归下降或自底向上分析程序而言，这类文法简单却不实用；然而，一个树文法并不受限于输入文件中单词的任何固有序序。

事实上，已有文献提出一些属性求值工具对属性文法执行详尽的分析，从而确定一个最佳的属性求值次序；由于这类分析的时间复杂度随文法的大小呈指数级增长，更可取的做法是留意到典型的编译程序设计人员早已认识到合适的属性求值次序，因而无需过度的负担即可告知编译程序生成工具。

代码清单 8.1 中的文法将其属性求值次序定义为二叉结点中从右到左的次序，这是由跟在树模板之后的语义动作中非终结符的引用次序规定的。在模板中以从左到右序列出各个子树并不会影响求值次序。

8.3.4 信息流与存储

代码清单 8.1 的例子使用了三种不同的信息管理方式，不幸的是它们在文献中全部被称为“属性”。首先，由于历史原因，我们使用了存储在符号表中的符号性质。尽管在经典的编译程序设计中，符号表是一个静态的数据结构，可根据当前的上下文进行伸展（和收缩），

而我们携带一个动态引用指向围绕着分析树的（可能是多个的）符号表。在这两种解释下，符号表组织和构造并无很大的差别。当符号表中的每一符号与一个或多个值相关联时，将由指向符号表的引用来访问这些值。这些值以前被称为符号的“属性”，但为避免混淆，本书自始至终地选择了术语“特性”（有时也称“值”）。代码清单 8.1 中的文法从符号表中抽取了两个特性：一个特性是布尔值，指明符号是否一个常量标识符；如果是，另一个特性是该常量的值。

其次，我们有文法产生式的继承属性和综合属性，这些术语被保留下来。在这一文法中，符号表是一个由非终结符 **Expn** 继承的属性，而符号表特性则被合成为综合属性；一个常量表达式子树的复合值随后被非终结符 **xform** 继承。

最后，与 TAG 一起新引入的还有内存中树数据结构的结点上附加的信息。一些文献将这种附加数据称为“属性”，而将这种结构本身称为“属性图”。本书为这些数据选择了更生动且不易混淆的术语“树的装饰”（或只称“装饰”），而将这种结构称为“带装饰的树”，尽管当我们提高结点的连通性以容纳全局优化所需的信息时，“树”这一术语将变得有些不准确。在这一文法例子中，一个标识符的名字和一个常量的值分别是它们各自结点的装饰。

一个“装饰”可以是任意的单个数据，它恰好与 AST 中的一个特定结点相关联。装饰可以是如上述例子所示的数值；该例子中装饰了两个不同的结点，其中一个装饰了一个常量的数值，另一个装饰了惟一的符号引用（如第 3 章所述，该引用通常是指向字符串表的一个下标）。装饰也可以是一个更复杂的数据对象，例如位的集合或甚至整棵子树。我们经常将一个指向树的另一部分的引用附加到一个结点上，从而以这种链接方式直接访问树的其他部分；如图 8-6 所示，其中 **call** 结点装饰了一个引用，该引用指向被调用过程的定义树。规定一个结点在任一时刻只限有一个装饰并没有什么道理，但却很实用；即便在这一严格规定之下，装饰也可设计为一个包含若干字段的记录。

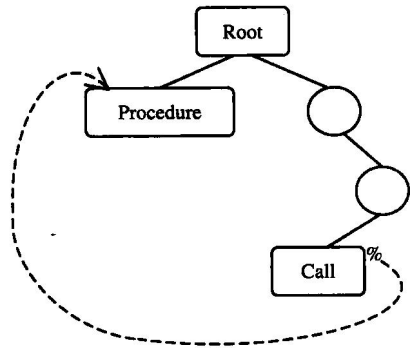


图 8-6 用指向另一子树的引用装饰一个树结点

8.3.5 带树值的属性

由于我们允许以一棵树的值作为树中结点的装饰，在文法中也可能有树值的属性。这成为一个强有力的工具，可将一个非局部优化变换涉及的所有位置收集到一两个非终结符的产生式中。

例如，一种理想的优化涉及将每次迭代都不会发生改变的所有计算移出循环之外。在 AST 中，典型情况是有一个结点指明了一个循环的头部（退出循环的测试和循环体则作为一个或多个特定的子树），而其循环体子树中可能存在只涉及循环不变量（真正的常量以及循环中不会改变的变量）的表达式子树，如图 8-7 所示。一种常见的有利做法是将这些循环不变表达式的子树从它们所处的树中剪除，然后嫁接到循环结点之外的树

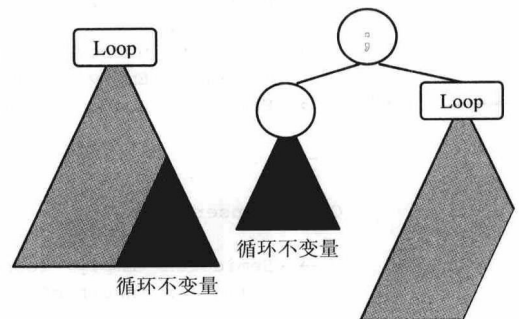


图 8-7 将循环不变代码移至循环体之外

上。借助于带树值的属性，我们有两种方法完成这一工作。

完成这一工作的最简单方式是遍历循环体，剪除循环不变表达式的子树，然后将它们作为综合属性沿树向上传递，如图 8-8 所示。这些被剪除出来的子树片断随后被嫁接到树遍历结束的地方。

stmt \downarrow inloop \uparrow movecode

- <Loop body>
- body: stmt \downarrow true \uparrow bodycon \uparrow bodycode
- ⇒ <Semicolon bodycode <Loop body>>
- <Semicolon notconst loopconst>
- [inloop = false; movecode = <Empty>]
- { 若不在循环之内则无变化 }
- <Semicolon notconst loopconst>
- [inloop = true; movecode = loopconst]
- ⇒ <Semicolon notconst <Empty>>

→ ...

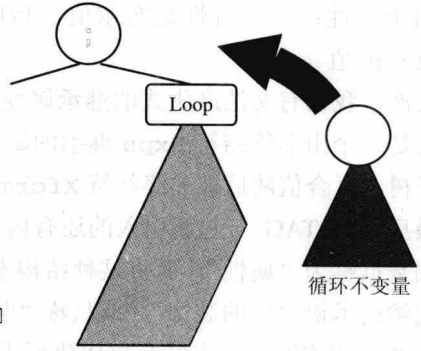
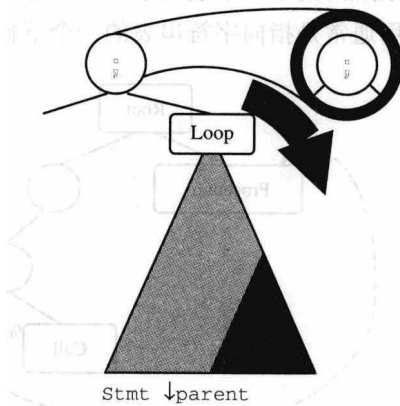


图 8-8 利用综合属性将循环不变代码移至循环体之外



- <Loop body>
- ⇒ newnode: <Semicolon moved: <Empty> <Loop body>>
- body: stmt \downarrow newnode
- <Semicolon notconst loopconst>
- [parent = <Empty>]
- { 若不在循环之内则无变化 }
- <Semicolon notconst loopconst>
- [inloop \neq <Empty>] parent: Graft \downarrow loopconst
- ⇒ <Semicolon notconst <Empty>>
- ...

Graft \downarrow insertcode

- <Semicolon <Empty> loopbody>
- ⇒ <Semicolon insertcode loopbody>

图 8-9 通过继承一个父结点的引用，将循环不变代码移出循环体之外

另一种方法是携带循环头部结点之上的结点的位置信息，作为一个继承属性向下传递到循环体中，类似一个向上回望父结点的舷窗，如图 8-9 所示。一旦识别到一个循环不变表达式，立即将它们从循环体中删除，并嫁接到这一父结点。由于该继承属性实际上是一个指向父结点的引用，在树的深层变换该属性相当于变换一个遥远的结点。在这个简单的示例中，仅有一个循环不变子树被提取出来；在实际应用中，多个剪枝可合并为单个子树一起移动。

8.3.6 不确定的分析

将上下文敏感文法用于编译程序设计的问题之一，是找不到一种算法构造确定的线性有界自动机。当越来越多的上下文敏感性通过属性引入上下文无关文法后，我们发现上下文敏感文法的不确定性亦随之而浮现。在图 8-8 和图 8-9 所示的代码移动例子中，两个文法均需要上下文信息以确定是否对树进行变换（以及将代码外提）。这两个文法所要求的上下文信息均是一对属性：一个继承属性用于确定外层上下文是否真的是一个循环，一个综合属性用于确定一个特定的子树是否真的是循环不变的（尽管这一点并不是那么明显）。

与我们之前一直使用的串文法相比，树文法本身就有些不确定性，因为当正在考虑的某一棵树有几个分枝时，在确定该树与源模板是否匹配之前就必须先检查这些分枝；如果其中任一分枝匹配失败，那么不管分析程序已沿树向下遍历了多深的层次，都必须退回原处，放弃这一产生式并尝试下一产生式。

一个变换文法有一个隐含的“其他”选项，在这种情况下将与任意未指定产生式的结点匹配，并将其变换为自身。通常设计一个树文法时，会显式地让不那么精确的模板去匹配那些更精确模板未能匹配的结点。例如，常量折叠变换文法可能有一条产生式检测含两个常量子树的表达式结点，如果这一检测失败则转而尝试检测单个常量的子树；如果首先尝试第二个选项，那么它肯定能够匹配，但我们的意图是尽可能匹配最精确的模板。这种启发式方法有时又称“最大咀嚼法”（这一说法可能源于小孩吃曲奇饼时喜欢最大限度地咬上一口），因为这种启发式方法倾向于选择最大的（因而也是最精确的）产生式模板。

8.4 组合串文法与树文法

如果编译程序使用树变换实现约束检查或代码优化，则分析程序必须将 AST 构造为它的语义动作。虽然近年已有研究将前端的分析与后端的功能合并为单个编译程序构造工具，但是大多数研究仍集中在如何无须集成即可解决其中的一些小问题。基于这一考虑，我们开辟一个新的话题，讨论适用于整个编译程序设计的统一语法结构。

有两种方法可将编译程序前端的串文法与在后端定义优化和代码生成的树文法联系在一起。最直接（在某种意义上也是最简单）的方法是使用一个翻译文法，其识别部分是一个串文法，其翻译部分则是一个树文法 [MetaWare, Inc., 1981]。

本书采用另一种方法，它将构造出来的中间代码树作为串文法中的综合属性。这一方法可显著提高编译程序设计的灵活性，并且因为我们出于其他理由已使用了属性文法，故不会带来额外的开销。此时，分析程序中的语义动作可包含构建树的成分，本质上就是使用第 5 章介绍的、现在应已熟悉的相同文法结构。与翻译文法相比，该方法的优点是编译程序构造工具更容易验证每一语言结构中的每一非终结符是否产生某种形式的中间代码树，以及它们是否正确地链接到一个完整的 AST 上。代码清单 8.2 比较了对同一表达式文法片段的两种不同处理方法。

代码清单 8.2 在分析程序中构造抽象语法树 (AST) 的两种方法。在 a 中, 源代码文本被“变换”为对应的 AST 结点; 在 b 中, 将 AST 结点构造为一个综合属性

Expn	Expn \uparrow outtree:tree
\rightarrow Expn "+" Term	\rightarrow Expn \uparrow etree "+" Term \uparrow ttree
\Rightarrow <Plus Expn Term>	[outtree = <Plus etree ttree>]
\rightarrow Term	\rightarrow Term \uparrow outtree
\Rightarrow Term	
a) 变换	b) 综合属性

如果按第 6 章的方法使用代码生成的语义动作, 那么根据一个 AST 生成代码非常类似于从分析程序生成代码。然而, 正如变换文法可用于构造一个 AST, 变换文法也可指定如何将这棵树平展为线性代码。现实计算机硬件的不规则性导致这一方法缺乏吸引力, 因而我们将代码生成工作局限于带语义动作的属性文法。

8.5 TAG 中的类型检查

由第 5 章可见, 约束检查既依赖于符号表中继承下来的上下文信息, 也依赖于在表达式叶结点综合得到的表达式类型属性。之前, 一个表达式运算符的操作数所要求的类型是根据语法来确定的 (**AND** 和 **OR** 的操作数总是布尔类型; **+**和*****的操作数总是整数类型), 但这并不是总能令人满意的。虽然 Ada 程序设计语言显式地允许程序员重载运算符, 但大多数程序设计语言只是重载了算术运算符, 使得它们在不同的数值类型 (**integer** 和 **real**) 上执行类似的功能。重载运算符的约束无法通过语法形式正确地检查, 但仍可在一个“一遍”分析程序中得到检查, 只要该语言禁止大多数的向前符号引用类别 (在 Ada 语言中这是成立的; **Pascal** 和 **Modula-2** 语言亦如此)。如果语言中没有要求一个符号“先声明、后使用”(很容易令人想起 **C** 和 **FORTRAN** 语言), 那么就不可能在一个“一遍”分析程序中实现强类型检查; 实际上, **C** 和 **FORTRAN** 这两种语言都不是强类型的。

如果将约束程序推迟到对程序的第二遍处理, 则“先声明、后使用”不再是强类型检查的必要条件。由于考虑到扫描和分析源代码文本文件的开销, “多遍”编译程序通常在 AST 上完成第二遍以及后续遍的处理, 而不是重新读入并分析源代码文件。对于一个约束程序而言, 树分析的不确定性受限于树文法的简单性; 约束检查通常并不需要在单个文法规则中有多个复杂的模板。如果约束能够在分析程序中得以检查, 那么就没什么理由再多用一“遍”来处理程序; 但如果语言的声明次序规则 (或因此而导致的次序不严格) 以及其他考虑因素强制这么做时, 这种做法可减少分析 AST 的开销。

如果不直接遵循第 5 章中展示的约束程序通用形式, 那么关于一个树文法中的约束检查也没有太多可深入讨论的。在典型情况下, 分析程序已构建了一个 AST, 其中的标识符叶结点已装饰了扫描程序返回的惟一符号下标; 该下标用于构建和查找符号表, 与之前的做法完全相同。约束程序对 AST 略加变换, 为运算符结点添加了类型装饰, 并且以适当的常量、变量或函数引用结点取代标识符结点; 在这种情况下, 这些新结点装饰的引用直接指向对应的变量或过程声明, 如图 8-10 所示。虽然 **Modula-2** 语言在语法上区别了函数调用和变量引用, 但 **Pascal** 和

一些其他语言却并非如此，因而约束程序必须适当地对树进行变换。当约束程序完成对程序树的变换后，符号表中的类型成分可以丢弃。

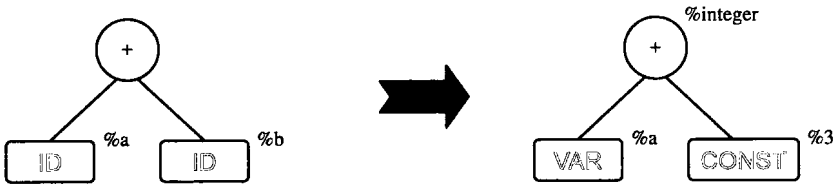


图 8-10 约束程序对表达式 $a + b$ 进行的树变换

一旦将语法制导的约束检查置之脑后，我们也可自由地将属性文法用于正确地强制规定变参的用法，而这在第 5、6 章强加的限制之下是做不到的。解决这类问题有两种通用的途径。最单纯的方法是定义一个分析程序的文法，使其所有的参数均接受表达式；然后，在约束程序中合成一个布尔类型的综合属性，该属性断言某一参数是否为单个变量引用；在所有情况下该属性都会被求值，但如果是值参则忽略该属性，且仅当符号表返回一个变参类型需匹配时才检查该属性。第二种方法允许以语义驱动一个分析程序，从而根据所需的是一个变参还是一个值参来决定应用不同的产生式。这种组合的方法更难以保障其正确性，但其优点是有益于代码生成的“一遍”编译。

8.6 基于变换的代码优化

上世纪 50 年代中期，最早的高级语言编译程序已提出十余种实用的代码优化变换 [Backus, 1981]。在随后的 35 年中，这一清单并没有增长很多，除了计算机体系结构的变化给代码改进带来新的机遇。因而，本小节只讨论经典的优化技术，说明如何在 TAG 中自然地定义每一种优化。

大多数代码优化的实现需要两个步骤。第一步是分析，收集信息以找出应用这一特定变换的机会；第二步才真正地对 AST 进行变换。这两步通常可在同一次遍历 AST 时完成；但同样也有许多时候，应用任何变换之前必须先收集相关信息。在后一种情况下，将多种优化变换的信息收集步骤合并为一次 AST 遍历会更加有益；类似地，多种树变换经常可合并为一个文法，表示在 AST 上的一次遍历。在 AST 上每一次不必要的分析或变换遍历意味着花费额外的编译时间，在编译程序的设计目标中当然希望尽可能让这些开销最少。

8.6.1 数据流分析

许多变换都需要使用一类称为“流分析”的分析技术，这种技术通常又称“数据流分析”(DFA)。数据流分析追踪整个编译单元（通常是过程）中的数据流向、变量用法或其他因素。某些数据流分析关注程序中向前流动的信息；其中一个例子是常量传播，即在程序执行路径中某一点的变量引用已知为某一常量值时，可用该常量取代这一变量引用。另一些数据流分析关注从程序结束处向后流动的信息；其中一个例子是消除那些对不再使用的变量的赋值。TAG 允许编译程序设计人员显式地指明分析的方向。

根据分析中所应用的是集合并运算（即“ \cup ”）还是集合交运算（即“ \cap ”），还可将数据流分析技术进一步细分。由于集合的并运算和交运算在数学上是对偶的，理论计算机科学家可能

比编译程序设计人员对这一划分更感兴趣。使用并运算和交运算均可定义大多数的数据流算法，取决于其中所涉及的集合属性的含义。

可证明，对某一特定类的数据流分析，集合属性定义了一个“格”，该格是值的偏序，使得并运算返回“大于”或等于其操作数的最小值，且交运算返回小于或等于其操作数的最大值。格的底元素是所有可能的值的交集，在数据流分析中表示“无信息”或“所有东西未定义”；而格的顶元素是所有可能的值的并集，可表示一个永不终止的假想程序。格点元素被赋予的特定含义取决于这些属性的目的，以及用于合并集合值的操作（并运算或交运算）。

数据流分析中最重要的数据结构是位向量，在 Pascal 和 Modula-2 语言中相当于一个集合。集合类型表示某一序数类型（通常是一个基数的子界，例如 $0..n-1$ ）的“幂集”，这一集合的值可以是 2^n 个包含 0 个或多个元素的可能类集之一，一个元素要么在该类集中，要么不在该类集中。

例如，设定子界为 $0..2$ ，则有 3 个集合元素：0、1 和 2，以及 8 个不同的可能集合： $\{\}$ 、 $\{0\}$ 、 $\{1\}$ 、 $\{2\}$ 、 $\{0,1\}$ 、 $\{1,2\}$ 、 $\{0,2\}$ 和 $\{0,1,2\}$ 。在这种情况下，一个集合在机器中很自然的表示方法是一个含 3 个位的位串，每一位的编号与子界基类型的元素匹配。整数 i 属于集合 s 当且仅当位串中的第 i 位为 1；空集表示为所有位为 0。对于小型集合而言，集合的交运算是大多数计算机硬件中的原语操作，即“逻辑与”运算符；一个元素属于两个集合的交集，当且仅当它同时是两个集合的成员。类似的，集合的并运算也是一个原语操作，使用“逻辑或”运算符；一个元素属于两个集合的并集，只要它属于两个集合中的某一个或同时属于两个集合。

许多数据流分析涉及变量的集合。每一变量被指定了一个惟一的序数，通常由约束程序将标识符添加到符号表时指定。这些序数未必是相邻的，但如果它们相邻则可令集合更加紧凑。然而，同一标识符表示的不同变量（可能位于不同的作用域中）必须具有不同的数字；我们关心的是真正的变量，而不是它们的名字。诸如数组和记录等聚合变量是一类特殊问题，稍后将详细讨论。

考虑代码清单 8.3 所示的一个简单程序例子。它有三个变量，我们随意编号为 $a=0$ 、 $b=1$ 和 $c=2$ ，从而它们可表示为一个集合中的相邻元素。在该程序上可执行两种不同的数据流分析，一个是向前的，另一个是向后的。

代码清单 8.3 用于展示数据流分析的小程序

```

MODULE DataFlow;
FROM IO IMPORT
    ReadInt, WriteInt;
VAR
    a, b, c: INTEGER;
1  BEGIN
2      a := 5;
3      ReadInt(b);
4      IF b = 3 THEN
5          c := a - b
6      ELSE
7          a := b;
8          b := 3
9      END;
10     WriteInt(c + b)
11 END DataFlow;

```

对于第一种分析，程序中传播的集合表示在每一点其值为已知常量的所有变量。如果程序执行到某一点时一个变量的值已知为常量，则该变量属于该集合。

初始时（代码清单 8.4 中的第 1 行），所有三个变量均未定义，因而集合为空（即位向量全部为 0）。执行第 2 行后，变量 **a** 具有常量值 5，因而集合为 { 0 }，其中仅包含变量 **a**。执行第 3 行后，变量 **b** 被定义，但其值来自编译时未知的输入数据，并不是一个已知的常量，因而变量 **b** 不会添加到集合中。然而在第 5 行开始时，变量 **b** 此时是已知的常量值 3——假如该变量不是 3 则会执行了 **ELSE** 路径，因而 **THEN** 部分开始时相关联的集合是 { 0, 1 }。由于 **a** 和 **b** 都是已知的常量，所以这两者之差也是常量，因而变量 **c** 将添加到集合中，在第 5 行结束时集合为 { 0, 1, 2 }。**ELSE** 部分（第 7 行）开始时的集合又是 { 0 }，因为此时关于变量 **b** 的所有已知信息只能保证 **b** 不等于 3，故 **b** 并不是集合中的成员。在这一行语句的执行过程中，变量 **a** 也不再具有一个已知的常量值，从而导致在该行结束后集合为空。然而在第 8 行，变量 **b** 被赋值为常量值 3，因而在 **ELSE** 部分结束时集合为 { 1 }。

代码清单 8.4 使用集合交运算的向前数据流分析

```
MODULE DataFlow;
FROM IO IMPORT
  ReadInt, WriteInt;
VAR
  a, b, c: INTEGER;
1 BEGIN
2     a := 5;           ----- { }
3     ReadInt(b);      ----- { 0 }
4     IF b = 3 THEN    ----- { 0 }
5         c := a - b;  ----- { 0, 1 }
6     ELSE              ----- { 0, 1, 2 }
7         a := b;      ----- { 0 }
8         b := 3;      ----- { }
9     END;              ----- { 1 }
10    WriteInt(c + b);  ----- { 0, 1, 2 } ∩ { 1 } = { 1 }
11 END DataFlow;
```

在到达第 9 行时，有两个候选的常数变量集合。从 **THEN** 部分出来的任何非常数变量在后续语句中肯定不是常量；类似地，**ELSE** 部分的任何非常数变量也须在后续语句中看作不是常量。因而，正确的集合应该是交集 $\{ 0, 1, 2 \} \cap \{ 1 \} = \{ 1 \}$ 。尽管变量 **b** 从两个条件分支出来都是常量，但这里 **b** 只是碰巧在两种情况下具有相同的值；一般情况下这并不成立，必须从已知为常量的变量集中取消该变量的资格。

在作为输出的表达式中，变量 **b** 是一个常量，但 **c** 却不是——实际上如果执行路径通过

ELSE 部分出来时, 变量 **c** 甚至是未定义的。如果我们以另一种方式定义集合, 那么在这里也许可以报告一个编译时错误, 即“变量 **c** 可能未定义”(参阅练习 4)。

为运用向后的数据流分析, 我们定义一个新的集合属性, 其中恰好包含了那些在变量中的值可能再次被使用的变量; 可能再次被使用的变量称为活跃变量。刚开始时, 惟一已知集合值的地方是在程序的结束处(代码清单 8.5 的第 10 行), 这里没有任何变量被使用, 因而集合为空且所有变量均为非活跃的。进入第 9 行后可知, 变量 **b** 和 **c** 在输出表达式中被使用, 因而将它们添加到活跃变量集中, 形成集合{ 1, 2 }。

代码清单 8.5 使用集合并运算的向后数据流分析

```

MODULE DataFlow;
FROM IO IMPORT
    ReadInt, WriteInt;
VAR
    a, b, c: INTEGER;
1 BEGIN
2     a := 5;
3     ReadInt(b);
4     IF b = 3 THEN
5         c := a - b;
6     ELSE
7         a := b;
8         b := 3;
9     END;
10    WriteInt(c + b);
11 END DataFlow;
```

----- { 2 }

----- { 0, 2 }

----- { 0, 1 } ∪ { 1, 2 } = { 0, 1, 2 }

----- { 0, 1 }

----- { 1, 2 }

----- { 1, 2 }

----- { 2 }

----- { 1, 2 }

----- { 1, 2 }

----- { }

该集合的值将以相同方式传播到两个条件支路。第 8 行的赋值语句取消(注销)了变量 **b** (在这一行之后, **b** 以前的值不再被使用); 但第 7 行注销变量 **a** 后, 变量 **b** 又被添加回集合中。注意, 此处关注的信息流是基于程序执行(运行时)的次序, 而不是文本的次序。因而, 对一个变量的赋值将先从正在处理的向后流动集合中注销该变量; 然后若在赋值给它的表达式中又使用了同一变量, 则令该变量再次复活。在 **ELSE** 部分开始处, 正在处理的集合是{ 1, 2 }。类似地, **THEN** 部分的赋值语句注销变量 **c**, 但将变量 **a** 和 **b** 添加到活跃变量集中。

当条件语句的两个支路在条件表达式测试处(即从执行次序看, 恰好在条件表达式求值之后)再次汇合, 我们取这两个集合的并集。在任一支路开始处活跃的变量, 必定在布尔表达式出来时也是活跃的, 从而我们有活跃变量集{ 0, 1 } ∪ { 1, 2 } = { 0, 1, 2 }。在 **IF** 语句中被测试的布尔表达式将变量 **b** 添加到该集合中, 但由于该变量本已在集合中, 故集合不作任何改变。

第 3 行的输入语句注销了变量 **b**；第 2 行的赋值语句注销了变量 **a**。

传播到第 1 行的活跃变量集本应为空集，表明这样一个事实：此时没有变量具有已定义的值可供使用；然而变量 **c** 却出现在活跃变量集中，这证明程序中存在一个缺陷，即我们之前提及的同一类错误。读者不妨自己在程序开头补回一条遗漏的对变量 **c** 的赋值语句，然后重做两种数据流分析，以验证这一修改确实消除了错误（参阅练习 5）。

8.6.2 数据流分析中使用属性文法

利用自己定义的集合属性类型以及适当的并运算和交运算属性求值函数，TAG 很自然地成为定义数据流分析的表示法。代码清单 8.6 展示了向后的活跃变量分析的文法片段。

代码清单 8.6 一个小型的数据流分析文法

LiveVars \downarrow inlive:set \uparrow outlive:set	
\rightarrow	$\langle \text{Assign } \langle \text{ID} \rangle \% \text{name expn} \rangle$ $[\text{exclude } \downarrow \text{name } \downarrow \text{inlive } \uparrow \text{asnset}]$ { 从传入集中删除名字 } $[\text{useless} = (\text{inlive} = \text{asnset})]$ { 标注该变量是否不在其中 } $\text{expn:LiveVars } \downarrow \text{asnset } \uparrow \text{outlive}$ { 将结果传递给表达式 } \Rightarrow $\langle \text{Assign } \langle \text{ID} \rangle \% \text{name expn} \% \text{useless} \rangle$ { 对结点加以装饰，以提示代码生成 }
\rightarrow	$\langle \text{ID} \% \text{name} \rangle$ $[\text{addset } \downarrow \text{inlive } \downarrow \text{name } \uparrow \text{outlive}]$ { 将名字添加到当前正处理的集合中 }
\rightarrow	$\langle \text{NUM} \% \text{value} \rangle$ $[\text{outlive} = \text{inlive}]$ { 忽略常量 }
\rightarrow	$\langle \text{Read } \langle \text{ID} \rangle \% \text{name} \rangle$ $[\text{exclude } \downarrow \text{name } \downarrow \text{inlive } \uparrow \text{outlive}]$ { 从传入集中删除名字 }
\rightarrow	$\langle \text{Write expn} \rangle$ $\text{expn:LiveVars } \downarrow \text{inlive } \uparrow \text{outlive}$ { 传递集合，使之流经表达式 }
\rightarrow	$\langle \text{Semi left rite} \rangle \mid \langle \text{Less left rite} \rangle \mid \langle \text{Equal left rite} \rangle \mid \langle \text{Grtr left rite} \rangle \mid$ $\langle \text{Plus left rite} \rangle \mid \langle \text{Minus left rite} \rangle \mid \langle \text{Star left rite} \rangle \mid \langle \text{Divd left rite} \rangle$ $\text{rite:LiveVars } \downarrow \text{inlive } \uparrow \text{midlive}$ { 使集合流经表达式和语句序列 } $\text{left:LiveVars } \downarrow \text{midlive } \uparrow \text{outlive}$ { 注意是从右到左流动 }
\rightarrow	$\langle \text{IF expn then else} \rangle$ $\text{then:LiveVars } \downarrow \text{inlive } \uparrow \text{thnlive}$ { 使集合流经 then 和 else 部分 } $\text{else:LiveVars } \downarrow \text{inlive } \uparrow \text{elslive}$ $[\text{union } \downarrow \text{thnlive } \downarrow \text{elslive } \uparrow \text{expnlive}]$ { 将其并集传递给表达式 } $\text{expn:LiveVars } \downarrow \text{expnlive } \uparrow \text{outlive}$

该文法中，单个非终结符 **LiveVars** 递归地遍历一棵程序树，以寻找变量引用和赋值语句。它有一个从右到左的属性，表示流经整个程序的集合。变量引用将它们各自的变量标识符编号（用 **ID** 结点上的装饰 **name** 表示）添加到集合中；赋值语句从集合中删除相应的元素。

一旦树遍历程序分析到一个赋值语句结点，变量将从传入的待处理集合中删除，即从产生的差集中删除该变量。如果结果集与传入的集合相同，则该标识符进入这一结点之时已经不活跃，因而该赋值语句没有作用；该结点将被附加一个布尔类型的标志作为装饰，从而让后续的

代码生成程序知道在这种情况下可消除该赋值语句。正处理的集合还将发送给整棵表达式子树，并且其结果将向上传递到树的左边。

当树遍历程序遇到一个标识符结点时，直接将其变量编号添加到传出的集合中；但是常量则对这一集合没有什么影响。**Read** 语句类似于无表达式的赋值语句，**Write** 语句令属性集流经右边的整棵表达式子树，而不作进一步处理。二元表达式的运算符子树和分号子树的遍历是从右到左，因而传入的待处理集首先被传递给右边的子树，然后其结果被传递给左边的子树。

上述文法中，条件语句是一个有趣的结点，因为 **then** 部分和 **else** 部分是并行的，而不是顺序的。这意味着同一个传入的位集被同时传递给两棵子树，其结果必须先采用集合的并运算组合在一起，然后再将结果传递给布尔表达式子树。

8.7 中间代码树表示的替代方案

将树作为程序的一种内部表示的关键优势之一，是可编写一个生成文法，使得程序中的每一结点恰好被访问一次（前序或后序），并采用传统的程序验证技术证明其正确性，从而更容易证明优化变换是完整的和正确的，这正是编译程序设计中的一个重要考虑因素。然而，程序树上的数据流分析要求源程序遵循结构化或无 **GOTO** 语句的程序设计风格。这是相对较新的程序设计语言进展，并非所有语言都满足这一约束。值得一提的是，就算是最近的某些程序设计语言也不是完全结构化的。例如，C 程序设计语言 **switch** 结构中的 **break** 语句在功能上与一条 **GOTO** 语句相同，因为它破坏了在每一结构中“单入口、单出口”的结构化风范。

鉴于编译程序设计人员始终有可能需要处理一些非结构化语言（其中最著名的是 C 和 FORTRAN 语言），很值得回顾一下更传统的中间代码表示方式，并讨论这类数据结构中的数据流考虑因素。通常这种格式称为“四元式”（有时也可能是“三元式”），这是基于以下事实：每一元素均由四个成分组成，包括一个运算符、一个目标操作数以及最多两个源操作数。

四元式表示法的灵感来源于大多数计算机的低级、非结构化体系结构。在这种体系结构中，控制流由一个运算或指令的序列决定，与第 6 章介绍的 Itty Bitty 栈机器非常相似。诸如 **while** 或 **if-then-else** 等高级语言结构将还原为条件分支和无条件分支；类似地，复杂表达式将还原为单运算的赋值语句序列，使用显式的临时变量保存中间结果值。代码清单 8.7 展示了代码清单 8.3 中样例程序的四元式。

代码清单 8.7 代码清单 8.3 中程序的四元式，同时展示了基本块

B1	a	←	5	
	t1	←	(CALL)	ReadInt

B2	b	←	t1	
	t2	←	b	= 3
	L1	←	(BRF)	t2

B3	t3	←	a	- b
	c	←	t3	
	L2	←	(BRA)	

B4	L1:	a	←	b
		b	←	3
	L2		←	(BRA)

B5	L2:	t4	←	c + b
			←	(CALL) WriteInt t4

优化一个以四元式表示的程序通常要求重新组织程序中的某些结构信息，而这些信息在生成四元式时通常被丢弃。程序员一般都会遵循结构化程序设计习惯，即便程序设计语言本身并没有支持或强迫他们这样做。虽然将一个程序翻译为四元式时删除了所有其他的结构，但已有大量研究试图通过对程序的数据流进行分析以恢复这些结构（或指出它们确实不存在）。

恢复程序结构的第 1 步是将线性的四元式串划分为“基本块”；这些基本块是四元式的序列，最多只包含一个标号（位于第一行），并且最多只有一条分支或调用指令（位于最后一行）。因而，只能从第一行进入一个基本块，并且只能在执行完最后一行后退出基本块。代码清单 8.7 中用虚线分隔了基本块。

下一步是构造一个有向图（而不是一棵树！），其中以基本块作为结点，分支指令和自然流向将这些结点连接在一起。图 8-11 展示了上述小例子中的基本块控制流图。

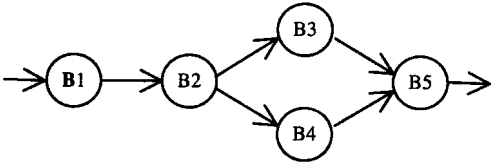


图 8-11 代码清单 8.3 中程序的控制流图

8.7.1 四元式的数据流

经典的数据流分析仅关注单个基本块中的数据流分析。Kildall 基于格模型将这种分析技术扩展到一个编译单元的整个图[Kildall, 1972]。图中的每一条边均附加了一个集合，初始时该集合为空，然后信息集向后流动（或向前，视具体情况而定），直至每一条边都已更新了新的集合值。若一个结点有两条或多条射入弧（例如本例中的 B5），则令其集合以相同方式流经每一射入弧；若一个结点有两条或多条射出弧（本例中的 B2），则将这些射出弧的集合的并集（或交集，视具体情况而定）传播给该结点。由于我们无法编写一个通用的 TAG 以最优次序（即便存在这样的次序）系统地访问控制流图中的每一结点，因而有必要迭代执行这一算法，直至所有集合稳定下来。

如果一个迭代的数据流算法在每次迭代中集合均始终如一地递增（附加在任何结点上的集合从不删除元素）或递减（从不添加新的元素），并且递增或递减的方向保持不变，则称该数据流算法为单调的。代码优化中使用的数据流分析算法通常是单调的。例如，代码清单 8.6 中的活跃变量分析是单调的；但常数变量的分析不是单调的，因为在程序某一特定点可见的一个变量可能在第一遍时被认为是一个常量，但稍后的分析（可能是一个循环的终止）在访问控制流图中一个结点的射出弧时，可能发现其求值表达式中某一分量有不同的值，而后续的迭代会考虑到这个表达式的值。

如果图中的分叉与汇合仅使用了集合的并运算（对于非递减的位向量）或交运算（对于非递增的向量），且集合的从属关系不依赖于集合中其他元素的从属关系，则很容易证明一个给定的算法是单调的。其他类型的算法也可能是单调的，但它们更加难以判定。可证明，一个单调的数据流分析算法总是可终止的，因为使用的是有穷集（典型情况是受到源程序中变量、表

达式、语句的数目限制)，且算法在所有集成为空集（交集的情况）或全集（并集的情况）之前将趋于稳定。

机灵的读者可能已注意到，有向程序图中的数据流分析相当于我们之前在程序树中对所有程序结构进行检查；然而我们强调在控制流图中必须进行迭代，对于树则一般没有强调。这里的本质区别在于非结构化的 **GOTO** 语句对控制流图带来的影响。图 8-12 展示了两个不可能以结构化程序结构表示的图。不存在算法可根据源代码构造这些图的树表示，并保证能够以一种正确的次序使得每一结点访问一次即可实现正确的数据流分析；实际上在其中的一种例子中，一次性访问的图遍历是肯定不够的。

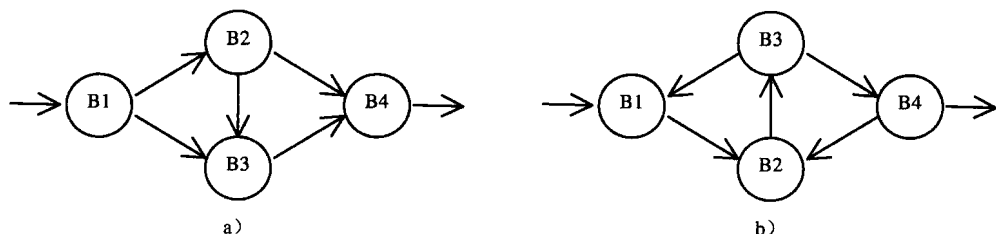


图 8-12 两个非结构化的控制流图。图 a 是 C 语言程序片段的一种简化形式；而图 b 只有在乱用 **GOTO** 语句时才可实现互锁的循环

8.7.2 循环的数据流分析

不同于条件语句的分叉（**if-then-else** 语句或 **case** 语句），循环结构的数据流分析通常无法在一遍中完成，其原因是：向前流过循环到达其出口的信息，必须在循环的入口处也是可用的；向后流过循环到达其入口的信息，必须在循环的出口处也是可用的。在基本块的有向图中，这一问题的显式解决方案是让数据流分析算法在整个图中迭代，直至最终趋于稳定。树表示中的数据流分析也可迭代直至稳定，但对于一个单调的算法，通常对每一循环体的处理最多只需两次：第一次遍历循环体就足以确定循环体中的语句对另一端集合的贡献；第二次遍历将该集合与流经循环出口的信息合并在一起（在向前数据流分析中）。

考虑代码清单 8.6 中的“引用一定值”数据流分析例子。由于数据向后流动，所以在循环出口处，活跃变量的组成是那些在循环入口处活跃的变量与跟在循环之后的代码中活跃的变量的并集。图 8-13 展示了一个小型循环完成了数据流分析后的集合。该程序片段有 3 个基本块，其中第 2 个基本块组成了循环体。在第 3 个基本块的入口处，活跃变量集包含元素 { **a**, **b** }。仅考虑循环体并忽略后续的两个块，第 2 个基本块入口处的活跃变量有 { **b**, **c**, **d** }，这意味着变量 **b**、**c** 和 **d** 在基本块中被注销之前会被引用。这两个集合的并集为 { **a**, **b**, **c**, **d** }，这正是流向循环出口的正确集合。在第二次遍历中，循环入口处出现相同的集合。注意，变量 **e** 在循环中首次被引用之前已被注销（即被赋值），因而在循环入口处该变量并非活跃的。

如果流经循环体的第一遍数据流分析携带了尽量多的信息，则另一端的集合在第一次遍历之后已经正确；只有循环体中使用的集合要求第二次遍历。引理 8.1 给出了更形式化的描述：

引理 8.1 一个单调的数据流分析算法用于一个仅含单个循环的结构化程序图时，在后续的迭代中无法令任何位于循环体之外的集合增大（其证明参阅练习 12）。

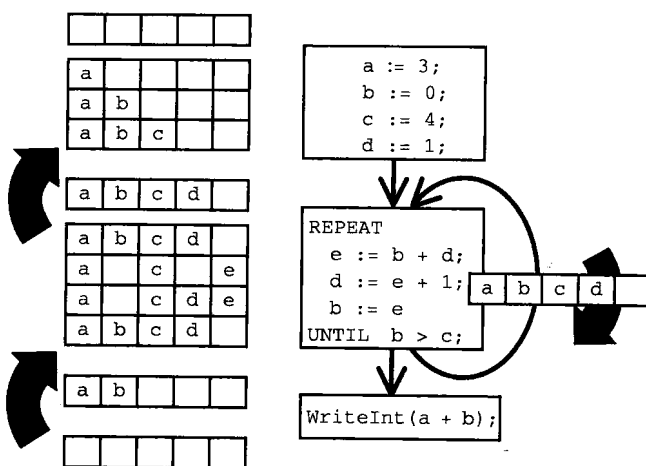


图 8-13 流经一个循环的“引用一定值”数据流分析

定理 8.1 对于一个单调的数据流分析算法，两次遍历一棵结构化程序的树就足够了。

定理 8.1 的证明可直接从引理 8.1 得到。我们应用此定理，对代码清单 8.6 的 TAG 进行扩展，以执行活跃变量分析，结果如代码清单 8.8 所示。注意，**WHILE** 和 **MODULE**（编译单元的根结点）这两个结点添加了新的产生式。我们从根结点出发，使用同一文法两次遍历程序树；与代码清单 8.6 相比，这里的惟一不同是对循环的处理。

代码清单 8.8 活跃变量分析的文法，其中包含 REPEAT 循环

LiveVars \downarrow pass:int \downarrow inlive:set \uparrow outlive:set

- <MODULE body>
body:LiveVars \downarrow 1 \downarrow empty \uparrow midlive { 第一次遍历程序 }
body:LiveVars \downarrow 2 \downarrow empty \uparrow outlive { 第二次遍历已是最终的结果 }
- <Assign <ID>%name expn>
[exclude \downarrow name \downarrow inlive \uparrow asnset] { 从传入集中删除名字 }
[useless = (inlive = asnset)] { 标注该变量是否不在其中 }
expn:LiveVars \downarrow asnset \uparrow outlive { 将结果传递给表达式 }
⇒ <Assign <ID>%name expn>%useless { 对结点加以装饰，以提示代码生成 }
- <ID>%name
[addset \downarrow inlive \downarrow name \uparrow outlive] { 将名字添加到当前正处理的集合中 }
- <NUM>%value
[outlive = inlive] { 忽略常量 }
- <Read <ID>%name>
[exclude \downarrow name \downarrow inlive \uparrow outlive] { 从传入集中删除名字 }
- <Write expn>
expn:LiveVars \downarrow pass \downarrow inlive \uparrow outlive { 传递集合，使之流经表达式 }
- <Semi left rite> | <Less left rite> | <Equal left rite> | <Grtr left rite> |
<Plus left rite> | <Minus left rite> | <Star left rite> | <Divd left rite>
rite:LiveVars \downarrow pass \downarrow inlive \uparrow midlive { 使集合流经表达式和语句序列 }

```

left:LiveVars ↓pass ↓midlive ↑outlive { 注意是从右到左流动 }

→ <IF expn then else>
  then:LiveVars ↓pass ↓inlive ↑thnlive { 使集合同时流经then和else部分 }
  else:LiveVars ↓pass ↓inlive ↑elslive
  [union ↓thnlive ↓elslive ↑expnlive] { 将其并集传递给表达式 }
  expn:LiveVars ↓pass ↓expnlive ↑outlive

→ <REPEAT body expn>%loopleftive
  ([pass = 1; exitlive = inlive] { 若第一次遍历,使用当前正处理的集合 }
  |[pass = 2; exitlive = looplive]) { 否则使用上一次遍历的结果 }
  [union ↓exitlive ↓inlive ↑expnlive] { 与传入集合并 }
  expn:LiveVars ↓pass ↓expnlive ↑bodlive
  body:LiveVars ↓pass ↓bodlive ↑outlive { 使集合流经循环体 }
⇒ <REPEAT expn body>%outlive { 对结点进行装饰,以备下一次遍历 }

```

REPEAT 结点的新产生式将流经循环体的位向量保存为树结点的一个装饰, 该结点的装饰可用于下一次遍历。在第一次遍历时, 该装饰并没有上一个副本可供使用, 因而取而代之的是到达循环出口处的待处理活跃变量集; 它将使用集合并运算与到达循环出口处的待处理集合合并在一起, 合并结果流向布尔控制表达式, 从而到达主循环体。活跃变量集流出循环体时, 将沿树向上向左传递; 同时保留一个副本作为该结点的装饰, 以供下次迭代时使用。

活跃变量分析并不是严格单调的, 因为注销变量时将从正在处理的集合中删除这些元素。这意味着在程序树上的两次遍历过程中, 对于包含两层嵌套的 **WHILE** 循环结构, 如果外层的控制表达式注销了一个变量 (可能是将它作为变参传递给某个函数, 并且已知该函数的副作用会改变它的值), 内循环的循环体将看到该变量仍是活跃的。这在安全性方面犯了错误, 并且仅当采用拙劣的程序设计风格时才有问题。很少实用的数据流分析算法是真正单调的, 但我们一旦意识到这一点, 即可有把握地忽略这类缺乏单调性的问题; 只是在非常罕见的情况下, 才会产生比最优情况略多的代码。

值得注意的是 **REPEAT** 循环并不会出现这一问题, 因为在循环体与控制表达式求值之间并不存在控制路径的中断。一个 **WHILE** 循环的基本块最小数目为 2, 如图 8-14 所示。我们将发现有几类优化工作在 **REPEAT** 循环上执行时, 略优于在 **WHILE** 循环上执行。奇特的是, 最初 FORTRAN 编译程序的 **DO** 循环拥有当时非常复杂的优化技术, 其机制更像 **REPEAT** 循环, 而不是 **WHILE** 循环。一个 **DO** 循环保证至少执行一次, 即使其控制变量的值域为空。

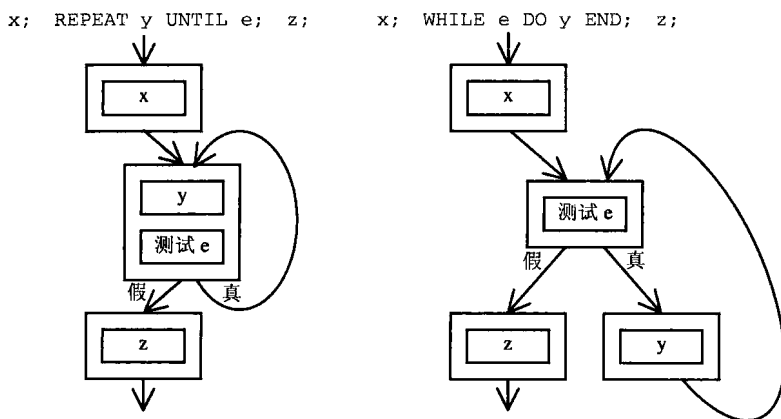


图 8-14 REPEAT 与 WHILE 循环在结构上的差异

8.8 实用优化变换综述

在深入掌握了分析与变换机制后，现在可讨论编译程序中常用的优化技术。本小节将集中讨论那些容易改写为 TAG 实现的优化，第 9 章再探讨某些需特别考虑的优化。

表 8-1 总结了在研究文献中已提出的 20 种不同的优化，并指出了每种优化所需的分析方法以及变换的类型。本质上有四类分析技术和三类通用的变换，后文将逐一讨论这些分析与变换。

表 8-1 20 种优化变换，同时列出了其分析与变换的类型。
表中的数字是指在本书第几章讨论了这些优化技术

第几章	优化技术名称	所需的分析	变 换
8	常量折叠	模拟执行	消除
8	消除死代码	模拟执行	消除
8	省略范围检查	模拟执行	消除
8	循环不变代码移动	循环结构、数据流分析	移动
9	归纳变量	循环结构、模拟执行	选择、消除
9	循环展开	循环结构、统计	移动（复制）
8	循环开关外提	循环结构	移动
9	循环融合	循环结构	移动
9	数组线性化	循环结构	消除
8	公共子表达式	模拟执行	消除
8	左、右提升	模拟执行、数据流分析	移动
8	复写传播	模拟执行	消除
9	寻址模式选择	统计	选择
9	分支链接	统计	选择
8	消除无用代码	数据流分析	消除
9	强度削弱	无	选择
9	资源分配	数据流分析、统计	选择
9	装入/存储优化	数据流分析	移动
8	数学等式	无	选择、消除
9	回代	统计	移动

本书已用不少篇幅讨论了数据流分析，这是代码优化中所需的最常见分析技术。数据流分析包括从左到右、或从右到左遍历程序树，并在分叉处可通过并运算或交运算积累集合数据。尽管这看起来像是优雅地划分了不同的种类，但还应注意某些形式的数据流分析必须扩展数据，这些数据只能与数学上的格相匹配。此外，由于并运算和交运算在数学上是对偶的，采用并运算的任何分析均可转换为采用交运算的互补术语。这个世界并不是总能够以优雅的数学范畴来划分的。

“模拟执行”有时又称“部分求值”，是指在编译时尽可能多地尝试执行一个程序，部分求值结果将会保存起来，以备在运行时完成求值。它实际上是一种向前数据流分析，不仅携带了集合，而且还包括变量和寄存器的内容、机器状态以及其他类型的信息。

“程序统计”关注在一个类似数据流的程序遍历中可收集的信息，但由于它是上下文无关的，所以并没有处理次序的限制。通常，它会计算某一上下文中标识符引用的个数，或估计一棵子树中生成的指令数目，诸如此类。

“循环结构分析”关注程序中控制循环的参数。大多数程序执行时间花费在程序的内循环，

因而已有大量研究深入探讨对循环结构的分析，以期达到改进其性能的目标。

“代码消除”是一种变换，通常是从程序树中剪除并丢弃多余的分枝。为在完成代码消除时仍能保持正确性，往往还需要对程序中的其他部分进行一些小变换。

“代码移动”是对程序树的一种变换，它将代码从程序经常执行的地方（例如循环）转移到不那么经常执行的地方；它也可以是将代码转移到离它被使用的位置更近的地方，从而可减少访问的开销。与此相关的优化还有复制代码的某一部分，以最大程度减少循环和过程中开销较大的判定操作。如前所述，TAG 中的代码移动由两个步骤组成：从程序中的某一部分剪除一棵子树，然后将它嫁接到另一部分。

“代码选择”与输出代码的选择密切相关。与其说代码选择是一种程序树的变换，倒不如说是为了准备最后一遍的树平展代码生成而装饰该程序树。

8.8.1 模拟执行优化的类别

表 8-1 中有六种优化技术将模拟执行列为其支撑分析技术。基于模拟执行的分析通常可与优化变换同时执行。模拟执行基本上按执行次序遍历一棵程序树，并且对循环和分叉予以特殊考虑。尽可能多地对每一条表达式进行“求值”，并“执行”每一条语句，从而在编译时确定寄存器和变量的内容等（如果无法确定实际的值，则用它们的信息来源表示）。

如果可以知道实际的值，那么这些寄存器或变量将是常量，因而可对树进行变换，以这些常量代替变量写入程序中；如果只能确定这些值的来源，那么模拟执行将寻找一些相似之处，这些相似之处意味着有机会保存并复用中间结果值。

常量值分析产生的结果是常量折叠和消除死代码，并且只要再多做一些工作，还可实现省略范围检查。公共子表达式分析产生的结果是消除公共子表达式、复写传播和左移动提升。

8.8.2 常量折叠分析

用于常量值分析的数据结构组成如下：每一待追踪的变量均有一个值字段，以及一个标志表明该值是否未知的；如果有可能，该标志也许是值字段中一个可区分的特殊值。通常这种分析仅限于标量变量，即整数类型、布尔类型、字符类型、枚举类型以及它们的子界类型等，但不会是集合、数组或记录类型。将实数类型包括在内是可能的，但不鼓励这样做，因为从事数值分析的人喜欢拥有对实数表达式求值的绝对控制。如果将信息量加倍，从而为每一标量变量保存其上界和下界，即可支持范围分析；但为保证正确地处理循环，推荐首先完成循环结构的分析。

从左到右遍历一棵程序树时，一个由所有变量的范围信息组成的从左到右属性将作为一个继承属性、然后作为一个综合属性在分析文法的每一条产生式中传播，与约束检查中传播符号表十分类似。实际上，符号表几乎就是实现这一数据结构的最佳方案，惟一区别是每一变量的值对在执行新的赋值时必须更新，并且访问变量的方式不是通过名字，而是通过它们惟一的变量编号。

除了在树中传播值集合之外，表达式子树还将合成当前表达式的范围（或值）。只要一个范围的上界与下界是一致的（即已知该值为一个常量），合成该值的表达式子树即可被剪除，代之以一个具有这个值的常量结点。这一剪枝动作通常会被推迟，直至该表达式用于一个更大的非常量表达式中，或赋值给一个变量，或用于其他场合（例如用作 **IF** 语句的控制表达式）。

当一条 **IF** 或 **CASE** 语句的控制表达式是一个常量时，就有可能不仅只剪除该子表达式的

子树，而且还可能剪除因为该常量值而不可达的分支情况的所有子树。这种优化技术称为消除死代码，因为这些不可达的代码树被认为是“死的”，意即它们无法被执行。实质上，消除死代码为程序员提供了一种免费的条件编译效果，因为如果包含在一条 **IF** 语句的 **THEN** 或 **ELSE** 部分之中，那么整块程序文本将不生成任何代码；这取决于一个具有合适值的常量，可能是一个全局常量。

在范围分析过程中遇到一个对标量变量的赋值时，该变量的范围值被设置为根据被求值的表达式推导出的范围（如果不是执行全面范围分析，则直接复制该值和标志）。通常，对一个标量变量的赋值将导致生成相应的范围检查代码，但是这一检测可在编译时的范围分析阶段完成；如果求值结果的范围安全地落在一个或两个边界中，那么这个赋值语句结点可用一个标志来装饰，该标志用于指示代码生成程序消除相应的测试与报错陷阱。数组的下标和 **CASE** 表达式的索引亦可采用相同的编译时检测。

代码清单 8.9 的文法片段展示了常量折叠与相关优化（省略范围检查以及消除死代码）的分析与变换的基本形式。这里假设分析程序或约束程序构造赋值语句结点时，有第三棵子树包含了范围检查的边界；该子树中的一个空结点表示代码生成程序不必执行检查。

代码清单 8.9 常量折叠的分析与变换文法

```

ConStmt ↓invars:table ↑outvars:table

→   <Assign <ID>%name expn bounds>
    expn:ConExpn ↓invars ↑evars ↑exlo ↑exhi   { 在表达式中寻找常量 }
    [replace ↓name ↓exlo ↓exhi ↓evars ↑outvars] { 更新值范围集合 }
    bounds:RngCheck ↓exlo ↓exhi               { 修改范围检查子树 }

→   <Read <ID>%name>
    [replace ↓name ↓-∞ ↓+∞ ↓invars ↑outvars] { 更新值范围集合 }

→   <Semi left rite>
    left:ConStmt ↓invars ↑midvars              { 令集合流经语句 }
    rite:ConStmt ↓midvars ↑outvars

→   this:<IF expn then else>
    expn:ConExpn ↓invars ↑evars ↑exlo ↑exhi   { 在表达式中寻找常量 }
    then:ConStmt ↓evars ↑thnvars              { 令集合流经then和else部分 }
    else:ConStmt ↓evars ↑elsvars
    ([exlo = 1; outvars = thnvars; nutree = then] { 为真则代之以 then 部分 }
    |[exhi = 0; outvars = elsvars; nutree = else] { 为假则代之以 else 部分 }
    |[otherwise; merge ↓thnvars ↓elsvars ↑outvars] { 否则，传递出其并集 }
    [nutree = this])
⇒   nutree ;

ConExpn ↓invars:table ↑outvars:table ↑loval:int ↑hival:int

→   <ID>%name
    [lookup ↓name ↓invars ↑loval ↑hival]      { 从当前正处理的集合取名字 }
    [outvars = invars]

→   <NUM>%value

```

```

[outvars = invars; loval = value; hival = value]    { 范围为常量 }

→  this:<Less left rite>
    left:ConExpn ↓invars ↑midvars ↑leftlo ↑lefthi { 令集合流经子表达式 }
    rite:ConExpn ↓midvars ↑outvars ↑ritelo ↑ritehi
    ([lefthi < ritelo; loval = 1; hival = 1; nutree = <NUM>%1]
      { 代之以 true, 合成 T,T }
    |[leftlo ≥ ritehi; loval = 0; hival = 0; nutree = <NUM>%0]
      { 代之以 false, 合成 F,F }
    |[otherwise; loval = 0; hival = 1; nutree = this])
      { 两者都不是, 合成 F,T }
⇒  nutree

→  this:<Equal left rite>
    left:ConExpn ↓invars ↑midvars ↑leftlo ↑lefthi
      { 令集合流经子表达式 }
    rite:ConExpn ↓midvars ↑outvars ↑ritelo ↑ritehi
    ([lefthi < ritelo; loval = 0; hival = 0; nutree = <NUM>%0]
      { 代之以 false, 合成 F,F }
    |[leftlo > ritehi; loval = 0; hival = 0; nutree = <NUM>%0]
    |[lefthi = ritelo & leftlo = ritehi; loval = 1; hival = 1; nutree = <NUM>%1]
      { 代之以 true, 合成 T,T }
    |[otherwise; loval = 0; hival = 1; nutree = this])
      { 两者都不是, 合成 F,T }
⇒  nutree

→  this:<Plus left rite>
    left:ConExpn ↓invars ↑midvars ↑leftlo ↑lefthi
      { 令集合流经子表达式 }
    rite:ConExpn ↓midvars ↑outvars ↑ritelo ↑ritehi
    [loval = leftlo + ritelo; hival = lefthi + ritehi]
      { 合成两者之和的范围 }
    ([loval = hival; nutree = <NUM>%loval]    { 如此则代之以常量结点 }
    |[otherwise; nutree = this])
⇒  nutree

→  this:<Minus left rite>
    left:ConExpn ↓invars ↑midvars ↑leftlo ↑lefthi
      { 令集合流经子表达式 }
    rite:ConExpn ↓midvars ↑outvars ↑ritelo ↑ritehi
    [loval = leftlo - ritehi; hival = lefthi - ritelo]
      { 合成两者之差的范围 }
    ([loval = hival; nutree = <NUM>%loval]    { 如此则代之以常量结点 }
    |[otherwise; nutree = this])
⇒  nutree

→  this:<Star left rite>
    left:ConExpn ↓invars ↑midvars ↑leftlo ↑lefthi
    rite:ConExpn ↓midvars ↑outvars ↑ritelo ↑ritehi
    [loval=min(leftlo*ritelo, leftlo*ritehi, lefthi*ritelo, lefthi*ritehi)]
    [hival=max(leftlo*ritelo, leftlo*ritehi, lefthi*ritelo, lefthi*ritehi)]
    ([loval = hival; nutree = <NUM>%loval]    { 如此则代之以常量结点 }

```

```

    | [otherwise; nutree = this])
⇒   nutree ;

```

RngCheck ↓exlo:int ↓exhi:int

```

⇒   <RngChk <NUM>%loval <NUM>%hival>
    [loval ≤ exlo & hival ≥ exhi]           { 在范围之内, ... }
⇒   <RngChk <> <>>                         { 故删除两个边界的检查 }

⇒   <RngChk <NUM>%loval <NUM>%hival>
    [loval ≤ exlo & hival < exhi]
⇒   <RngChk <> <NUM>%hival>                 { 删除下界的检查 }

⇒   <RngChk <NUM>%loval <NUM>%hival>
    [loval > exlo & hival ≥ exhi]
⇒   <RngChk <NUM>%loval <>>                 { 删除上界的检查 }

⇒   <RngChk <NUM>%loval <NUM>%hival>
    [loval > exlo & hival < exhi]           { 不知是否在范围中, }
⇒   <> ;                                     { 故不消除任何范围检查 }

```

内置属性求值函数 **replace** 创建一个新的值表，其中某一特定变量的范围被两个给定的值取代。注意实现该函数时必须小心，因为旧的表也可能沿分叉的其他分枝向下传递并独立地更新。因而，这里很重要的一点是应将属性保存为一个值，而不是一个可能被改变的数据结构。

属性求值函数 **merge** 接受两个范围表（通常派生自单个父结点），并创建一个新表，其中每一变量的范围是两个输入表中相应范围的合并。因而对每一变量而言， $LoOut = \min(LoIn1, LoIn2)$ 且 $HiOut = \max(HiIn1, HiIn2)$ 。这些求值函数的实现留待读者自己完成。应注意到格的底范围不包含任何值（可表示为 $[-\infty, +\infty]$ 或 $[1, 0]$ ，或任意其他空范围），在分析开始时将用于初始化所有变量的表。另一种做法是，如果分析从一个空表开始，底范围可作为一个特殊的值，表示变量仍未加入到表中。

还有一种实用的变换在代码清单 8.9 中未展示出来，即查找那些带有不相邻常量子树的双运算符的树片段，如图 8-15 中的例子所示。尽管将累加的常量向上移至表达式树的根结点并未直接引起代码的简化或约简，但其结果是这种形式的表达式树在编译程序生成的下标表达式中很常见，它的最后一个操作是一个内存引用。大多数计算机支持一种索引偏移量寻址模式，该寻址模式实际上是让一个几乎没有什么开销的常量加法作为地址表达式计算的最后操作；因而在这种情况下，将常量之和的项传播到表达式树的根结点可产生更快、更紧凑的代码。

代码清单 8.9 中的文法片段也未展示循环中的常量折叠。类似于数据流分析，只需两次遍历就足以检测出常量表达式和变量，但是全面范围检查还需要一些关于循环结构的信息；稍后在讨论此类优化时，我们再更深入地探讨这一课题。读者不妨自己修改该文法以消除范围检查，并正确地处理循环中的常量表达式。

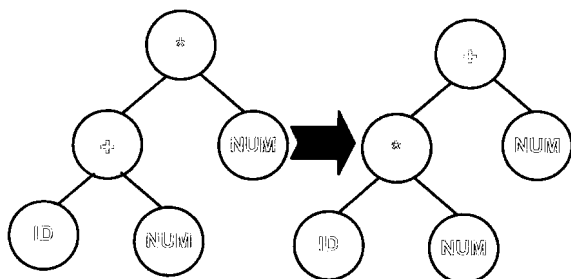


图 8-15 将累加的常量移至表达式树的根结点

8.8.3 使用值编号检测公共子表达式

与常量折叠类似，消除公共子表达式（CSE）时在程序树中传播的数据结构也是一个符号表。然而在这种情况下，用于访问表中一个入口的“名字”是一个子表达式结点，而不只是一个变量的编号。这样的名字在表中的“值”是一个对计算该子表达式的子树的引用。当对一棵树进行遍历以检查子表达式时，会在表中查找每一结点；如果找到，则将树变换为使用一个存储了该值的临时变量。

这里的算法是首先由 J. Cocke 和 J.T. Schwartz 于 1970 年发表的“值编号”系统的一种简化版。他们先对值进行编号，然后为该表达式构造一个有向无环图（DAG），其中直接使用了一个指向原有子表达式树的引用（指针）。另一些研究人员建议采用一种“可用表达式”的数据流分析技术 [Aho&Ullman, 1973]；但本质上两者是相同的计算，只是后者的表述方法不够浅显易懂。

子表达式结点的信息作为一个标识符登记到表中，它由一个运算符（结点名字）及其操作数的值编号组成。因而一个二元运算符结点包括三个部分，即便采用了数值类型的值编号，产生的记录依然太大，不利于方便、快捷的表查找操作。为取得合理的符号表查找性能，推荐对子表达式的特征进行散列（参阅第 3 章关于散列表技术的讨论）。利用一个合适的大型散列表，查找时间可减少到每次访问仅需要一次或两次比较运算。散列表中活跃的入口通常由可见的变量以及不超过几十个的可用表达式组成，因而将表的大小设置为 100~500 已经非常充足，特别是在诸如 Modula-2 这类限制了非局部标识符可见性的语言中。

商用编译程序通常不考虑全局消除公共子表达式，因为一般认为这类优化需要在整个程序中迭代执行数据流分析。本书采用一种单调的算法，从而将程序图限制为树，只用两次遍历即可取得优化的代码。然而可证明，第一次遍历只需找出循环中被修改的变量，而模拟执行仅用一次遍历即可完成。

为展示消除公共子表达式的算法是如何工作的，考虑图 8-16 所示的程序树。树中的结点采用罗马数字编号，对应于表 8-2 中的值表序列。值编号作为属性沿树向上传递，在图中用空心阿拉伯数字表示。从顶部开始遍历该树时，表达式表为空（结点 i）；另一种做法是将该表初始化为包含全部活跃变量，且将所有值编号均初始化为只是指向自身的引用。

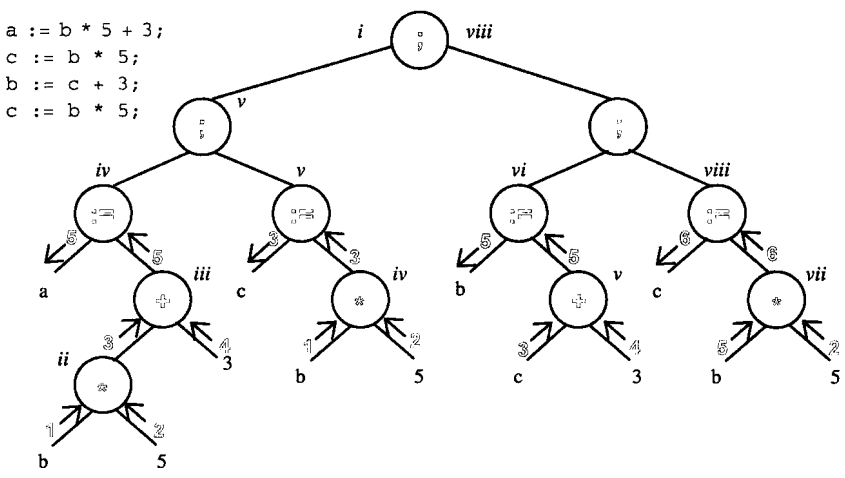


图 8-16 消除公共子表达式示例。小罗马数字对应于表 8-2 中的快照；空心数字是值编号（详细解释请参阅本书相应内容）

表 8-2 图 8-16 的值编号表序列

i	(空)	
ii	b	1
	5	2
	*, 1, 2	3
iii	b	1
	5	2
	*, 1, 2	3
	3	4
	+, 3, 4	5
iv	b	1
	5	2
	*, 1, 2	3
	3	4
	+, 3, 4	5
	a	5
v	b	1
	5	2
	*, 1, 2	3
	3	4
	+, 3, 4	5
	a	5
	c	3
vi	b	5
	5	2
	*, 1, 2	3
	3	4
	+, 3, 4	5
	a	5
vii	c	3
	b	5
	5	2
	*, 1, 2	3
	3	4
	+, 3, 4	5
	a	5
viii	c	3
	*, 5, 2	6
	b	5
	5	2
	*, 1, 2	3
	3	4
viii	+, 3, 4	5
	a	5
	c	6
	*, 5, 2	6
	b	5
	5	2

首先遇到的稍复杂结点是一条赋值语句。按深度优先的模拟执行次序向下访问其表达式子树，将到达标识符结点 **b** 并返回值编号 1，指向变量 **b** 中的这一特定的值（并不是 **b** 中的任意

值，而是在执行过程中这一时刻 **b** 所包含的值)。乘法结点的右子表达式中，常量 5 返回的值编号为 2；此时乘法结点有一个复合值 $(*, 1, 2)$ ，该值不在表中；它将作为值编号 3 登记到表中（结点 *ii*），并且该值编号沿树向上传递给加法结点。值编号 3（刚刚形成的）和值编号 4（常量 3）以类似的方式相加，为复合表达式 $(+, 3, 4)$ 产生一个新的值编号 5。赋值运算符为变量 **a** 将一个新的值登记到表中，该值即是相同的值编号 5（结点 *iv*）。

上述例子的第二条赋值语句中，变量 **b** 已在表中且值编号为 1，故不必修改符号表即可使用该值；类似地，常量 5 返回的值编号为 2。乘法结点合成一个表达式 $(*, 1, 2)$ ，并发现该表达式也已在表中，且值编号为 3。该值再次沿树向上传递给赋值运算符；这相当于在程序运行时若到达了程序中的这一位置，则已计算出积 $b * 5$ 。变量 **c** 以值编号 3 登记到表中（结点 *v*）。为这第二个乘法生成的实际代码通常只是一个寄存器的存储操作，该寄存器包含了已保存的中间值。在第三条赋值语句中，变量 **c** 返回值编号为 3，并且在查表时表达式 $(+, 3, 4)$ 立即返回编号 5；这一值编号取代了变量 **b** 先前的值（结点 *vi*）。

在该例的最后一条赋值语句中，又遇到子表达式 $b * 5$ ，但变量 **b** 中的值已改变。因而，在表中查找复合表达式 $(*, 5, 2)$ 时，该表达式不存在，故必须添加到表中（结点 *vii*）；这样得出的一个新的值编号 6 成为变量 **c** 的值，取代了变量 **c** 先前的内容（结点 *viii*）。

当分析进行到这一位置时，值编号 3 是不可访问的，因为其名字为值编号 1 和 2 的积，而值编号 1 根本就已不在表中。如果只是在一个基本块中运用消除公共子表达式的算法，这些碎片几乎没有积累的机会，并占用过量的编译时存储空间。但在全局子表达式分析中，有希望以这一方式消除已被注销的值编号。只要一个变量被赋予一个新的值编号，在其名字中包含旧值的任何其他值编号也可被清除。这可以递归地应用（尽管上述例子并不属于这种情况），譬如这些刚死去的值编号还将导致消除以这些值编号部分命名的其他值编号。

代码清单 8.10 中的文法片段展示了消除公共子表达式的分析与变换的基本形式。新的内置属性求值函数 **compose** 接受一个结点名字（表示为一个无任何子树的结点构造算子）和两个值编号，返回一个可用于在符号表中查找该表达式的新的值编号名字。该函数的工作原理实际上与第 3 章介绍的字符串表机制相似；其实现留待读者完成。代码清单 8.10 中的文法片段仅展示了两个表达式结点；除结点名字之外，所有表达式的处理是相同的。

代码清单 8.10 消除公共子表达式的文法片段

```
CSEStmt ↓invals:table ↑outvals:table

→   <Assign <ID>%name expn>
      expn:CSEExpn ↓invals ↑evals ↑valtree      { 取表达式的值编号 }
      [replace ↓name ↓valtree ↓evals ↑outvals]  { 更新值编号表 }
⇒   <Assign <ID>%name valtree>                  { 用新的树取代表达式树 }

→   unique:<Read <ID>%name>
      [replace ↓name ↓unique ↓invals ↑outvals]  { 给它一个新的值编号 }

→   <Semi left rite>
      left:CSEStmt ↓invals ↑midvals              { 令集合流经语句 }
      rite:CSEStmt ↓midvals ↑outvals
```

```

→ <IF expn then else>
   expn:CSEExpn ↓invals ↑evals ↑valtree { 取表达式的值编号 }
   then:CSEStmt ↓evals ↑thnvals { 令集合流经 then 和 else 部分 }
   else:CSEStmt ↓evals ↑elsvals
   [mergevals ↓thnvals ↓elsvals ↑outvals] { 传递出其并集 }
⇒ <IF valtree then else> ; { 用新的树取代表达式树 }

```

CSEExpn ↓invals:table ↑outvals:table ↑valtree:tree

```

→ valtree:<ID>%name
   [compose ↓<ID> ↓name ↓0 ↑valno] { 计算一个值编号 }
   [lookup ↓valno ↓invals ↑tableval] { 在表中查找该值编号 }
   ([tableval = <>; valtree = vartree])
   [replace ↓name ↓vartree ↓invals ↑outvals] { 如果不存在则加入表中 }
   |[otherwise; outvals = invals; valtree = tableval])

→ contree:<NUM>%value
   [compose ↓<NUM> ↓value ↓0 ↑valno] { 计算一个值编号 }
   [lookup ↓valno ↓invals ↑tableval] { 在表中查找该值编号 }
   ([tableval = <>; valtree = contree])
   [replace ↓name ↓contree ↓invals ↑outvals] { 如果不存在则加入表中 }
   |[otherwise; outvals = invals; valtree = tableval])

→ exptree:<Less left rite>
   left:CSEExpn ↓invals ↑midvals ↑leftval { 令集合流经子表达式 }
   rite:CSEExpn ↓midvals ↑expvals ↑riteval
   [compose ↓<Less> ↓leftval ↓riteval ↑valno] { 计算一个值编号 }
   [lookup ↓valno ↓invals ↑tableval] { 在表中查找该值编号 }
   ([tableval = <>; valtree = exptree])
   [replace ↓name ↓exptree ↓expvals ↑outvals] { 如果不存在则加入表中 }
   |[otherwise; outvals = expvals; valtree = tableval])
⇒ <Less leftval riteval> { 取代子表达式树 }

→ exptree:<Plus left rite>
   left:CSEExpn ↓invals ↑midvals ↑leftval { 令集合流经子表达式 }
   rite:CSEExpn ↓midvals ↑expvals ↑riteval
   [compose ↓<Plus> ↓leftval ↓riteval ↑valno] { 计算一个值编号 }
   [lookup ↓valno ↓invals ↑tableval] { 在表中查找该值编号 }
   ([tableval = <>; valtree = exptree])
   [replace ↓name ↓exptree ↓expvals ↑outvals] { 如果不存在则加入表中 }
   |[otherwise; outvals = expvals; valtree = tableval])
⇒ <Plus leftval riteval> ; { 取代子表达式树 }

```

所谓“复写传播”是指这样的一种优化：找出将同一个值赋值给几个变量的多条赋值语句，并消除不必要的寄存器装入运算。它节省的代码或执行时间非常少，但与消除公共子表达式一起执行时，这一优化实际上没有什么开销。

8.8.4 左移动提升

所谓“提升”是指这样的一种优化：寻找一个分叉（**IF-THEN-ELSE** 语句，或 **CASE** 语句的所有支路）的所有支路中的相同代码，并将这些代码移至分叉之前或之后的公共代码中。一个分叉的每一支路开始处的相同代码可向前（向左）移至刚好在条件分支之前；每一支路结束

处的相同代码可向后（向右）移至支路的汇合点。一般来说，提升并不会使代码运行得更快，但是当空间非常珍贵时，它可节省代码的空间。

左移动提升的分析过程与消除公共子表达式非常类似。在程序中消除公共子表达式的同时，为一个分叉的 **THEN** 部分保存一个表达式表；当处理 **ELSE** 部分时，每一个新的表达式都需要查找该表（但即便不存在也不会将表达式添加到表中）。在分叉的另一支路的表中找到的所有表达式均可作为候选的提升代码，应从每一支路中剪除这些代码，并嫁接到该分叉的判定代码之前的公共代码结束处。

一种简单但速度不是特别快的左提升技术是两次运行消除公共子表达式的算法，第一次先将被提升的候选代码副本嫁接到公共代码，第二遍则由普通的消除公共子表达式算法将原枝剪除。该方法无法正确地提升对一个带副作用的过程的相同调用；但除此之外，它与仅用一次遍历的、更直接的剪枝和嫁接方法同样高效。由于这一技术与传统的消除公共子表达式算法差别甚小，其实现细节留待读者完成。



8.8.5 右移动提升

在中间代码的树表示中，执行右移动提升比左提升稍复杂一些。考虑以下程序片段：

```
IF x THEN
    z := a + b
ELSE
    z := a * b
END
```

上例中可提升的公共代码是对变量 **z** 的赋值。表 8-3 展示了 Itty Bitty 栈机器（参阅附录 C 了解如何转换为零地址）的一种单地址变形的代码，其中提升只是一个简单的变换。然而在树表示中，树的变换就远远没有那么简单了，如图 8-17 所示。

表 8-3 机器代码的右移动提升

未优化的原代码		右提升之后	
	LD x		LD x
	BRF else		BRF else
	LD a		LD a
	ADD b		ADD b
	ST z		{被移走}
	BR join		BR join
else	LD a	else	LD a
	MPY b		MPY b
	ST z		{被移走}
join	...	join	ST z {被提升的代码}
			...

这一问题的根源在于程序执行时是自底向上遍历一棵树，因而左提升可从树的末梢收集叶结点和小分枝，在这些地方容易将它们剪枝，然后嫁接到其他子树的末梢。由于右提升涉及一棵子树中最后执行的操作，提升的候选者往往是内部结点，而叶结点和小分枝必须以某种方式驻留原处！只有对树进行彻底的变换，方可实现这种形式的提升，正如图 8-17 所作的变换：有两个赋值语句子树的条件语句必须变成单条赋值语句，并以一个条件表达式作为被赋予的值。

在实际应用中，我们会创建新的临时变量以保存这些中间值。

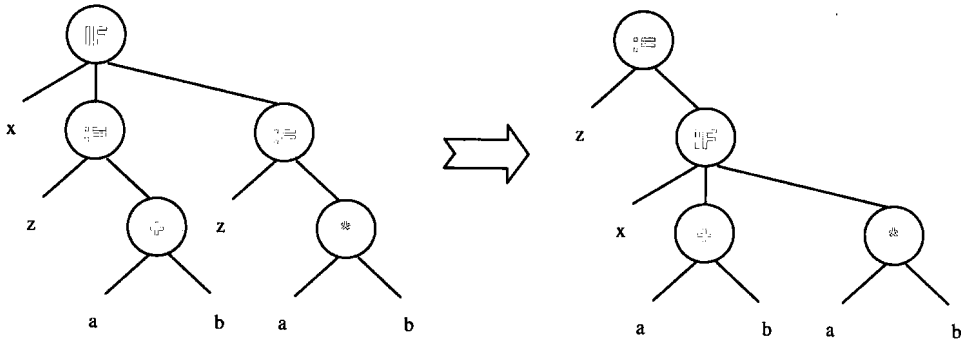


图 8-17 树表示中的右移动提升问题

代码清单 8.11 是一个右移动提升的文法片段，其核心算法在于将两块代码树片段沿程序树向上流动，其中一块是待提升的代码集，另一块是驻留原处的代码集。这两块代码集中，有一块将作为每一产生式中变换后的结点，另一块则作为综合属性向上传递。至于哪一块作为属性，哪一块作为变换，取决于当前结点是被提升还是驻留原处：若当前结点拟驻留原处，则拟驻留的代码块将继续作为当前结点的子树；若拟提升当前结点，则拟提升的代码块将添加到当前结点作为一棵子树，并将驻留原处的代码作为属性向上传递。

代码清单 8.11 右移动提升的文法片段

```

RHoist  ↓invals:table ↓invref:set ↓invasn:set ↓lastleg:bool ↓hoistme:bool
        ↓parentval:tree
        ↑outvals:table ↑outvref:set ↑outvasn:set ↑detached:tree

→      myself:<IF expn then else>                { 待提升的树的根结点 }
        else:RHoist ↓empty ↓empty ↓empty ↓false ↓false ↓<>
              ↑elsvals ↑elsvref ↑elsvasn ↑elsdet    { 令集合流经两个支路 }
        then:RHoist ↓elsvals ↓empty ↓empty ↓true ↓true ↓<>
              ↑thnvals ↑thnvref ↑thnvasn ↑thndet
        [union ↓thnvref ↓elsvref ↑midvref]          { 合并表达式的集合 }
        [union ↓thnvasn ↓elsvasn ↑midvasn]          { 此处并非完全递归 }
        expn:RHoist ↓invals ↓midvref ↓midvasn ↓lastleg ↓false ↓myself
              ↑outvals ↑outvref ↑outvasn ↑detached
⇒      <Semi <IF expn thndet else> then>          { 执行树的变换 }

→      <Semi left rite>
        rite:RHoist ↓invals ↓invref ↓invasn ↓lastleg ↓lastleg ↓<>
              ↑midvals ↑midvref ↑midvasn ↑ritedet    { 令集合从右到左流动 }
        left:RHoist ↓midvals ↓midvref ↓midvasn ↓lastleg ↓lastleg ↓<>
              ↑outvals ↑outvref ↑outvasn ↑leftdet
        [detached = <Semi leftdet ritedet>]        { 收集剪下的部分 }

→      myself:<Assign <ID>%name expn>
        ([lastleg = false; valtree = myself; hoistself = false] { 第一个支路 ... }
        |[inset ↓name ↓invasn; midvals = invals] { 无法在赋值语句上提升 }
        |[inset ↓name ↓invref; midvals = invals] { 无法在变量引用上提升 }
        |[otherwise; into ↓myself ↓invals ↑midvals] { OK, 将自身放入表中 }

```

```

| [otherwise] { 第二个支路 ... }
  [compose ↓<Assign> ↓<ID>%name ↑valno]
  [lookup ↓valno ↓invals ↑tableval; midvals = invals] { 在表中查找自身 }
  ([tableval = <>; valtree = myself; hoistself = false]
  |[inset ↓name ↓invasn; valtree = myself; hoistself = false]
  |[inset ↓name ↓invref; valtree = myself; hoistself = false]
  |[otherwise; valtree = tableval; hoistself = true]
    { 找到; 提升自身 }
  tableval:Prune ↓0)) { 同时剪除另一支路 }
expn:RHoist ↓midvals ↓invref ↓invasn ↓lastleg ↓hoistself ↓valtree
  ↑outvals ↑outvref ↑midvasn ↑expdet
[addset ↓name ↓midvasn ↑outvasn] { 注意该赋值语句 }
([lastleg = hoistself; detached = expdet; xfrm = myself]
  { 子结点跟随其父结点 }
|[otherwise; detached = myself; xfrm = expdet]) { 父、子结点分道扬镳 }
⇒ xfrm { 向上发送新的子结点 }

→ exptree:<Plus left rite>
  ([lastleg = false; valtree = exptree; hoistself = false]
    { 第一个支路 ... }
  [into ↓exptree ↓invals ↑expvals] { 将自身放入表中 }
  |[otherwise] { 第二个支路 ... }
  [compose ↓<Plus> ↓parentval ↑valno]
  [lookup ↓valno ↓invals ↑tableval; expvals = invals]
    { 在表中查找自身 }
  ([tableval = <>; valtree = exptree; hoistself = false]
  |[hoistme = false; valtree = exptree; hoistself = false]
  |[otherwise; valtree = tableval; hoistself = true]
    { 找到; 提升自身 }
  tableval:Prune ↓0)) { 同时剪除另一支路 }
rite:RHoist ↓expvals ↓invref ↓invasn ↓lastleg ↓hoistself ↓valtree
  ↑midvals ↑midvref ↑midvasn ↑ritedet
left:RHoist ↓midvals ↓midvref ↓midvasn ↓lastleg ↓hoistself ↓valtree
  ↑outvals ↑outvref ↑outvasn ↑leftdet { 令集合流经子表达式 }
exptree:PTangle ↓leftdet ↓ritedet ↓hoistme ↓hoistself ↓tableval ↑detached

→ vartree:<ID>%name
  [inset ↓name ↓invasn; hoistme = true] { 无法在赋值语句上提升 }
  [tempvar ↑varno] { 创建一个新的临时变量 }
  [detached = <ID>%varno]
⇒ <Assign <ID>%varno vartree> { 在临时变量保存变量值 }

→ vartree:<ID>%name
  [otherwise; detached = <>] { 否则就跟父结点一样 }
  [outvals = invals; outvref = invref; outvasn = invasn] ;

PTangle ↓leftdet:tree ↓ritedet:tree ↓hoistme:bool ↓hoistself:bool ↓oldtree:tree
  ↑detached:tree

→ exptree:<Plus left rite>
  [hoistme = hoistself] { 子结点跟随其父结点 }
  [detached = <Semi leftdet ritedet>] { 收集剪下的部分 }

```

```

→   exptree:<Plus left rite>
      [otherwise; tempvar ↑varno]                { 父、子结点分道扬镳 }
      [detached = <Semi <Assign <ID>%varno exptree> <Semi leftdet ritedet>>]
      oldtree:Prune ↓varno                        { 另一支路中的同一变量 }
⇒   <ID>%varno ;                                { 临时变量携带该值 }

```

Prune ↓varno:int

```

→   <Plus left rite> | <Assign left rite>
      [varno = 0]                                { 没有值要保存到变量中 }
⇒   <Semi left rite>

→   exptree:<Plus left rite>
      [varno > 0]                                { 将部分值保存到变量中 }
⇒   <Assign <ID>%varno <Plus left rite>> ;

```

对于每一结点，对它进行提升亦或令其驻留原地的决策仅取决于其自身的结构以及继承得到的信息。其中，继承信息是一个布尔值，表明其父结点是否将被提升。基于这一知识，一个结点可确定如何定位向上传递给父结点的两块子树。一共有四种可能的情况，其中的两种是一个结点及其父结点一起提升或驻留原处，另外两种是它们的处理分道扬镳。由于是否提升的决策取决于父结点是否也被提升，上述情况中有一种是不存在的。然而，一个父结点可以被提升，而令其子结点驻留原处而不提升；在这种情况下，两块子树在搬运过程中被交换，从而应放在一起的代码最终会放在一起。图 8-18 以图形方式展示了这一决策过程，注意如果将一个子树结点提升到右边，而令一个依赖于该子树的值的父结点驻留原处，这会破坏程序原有的语义。

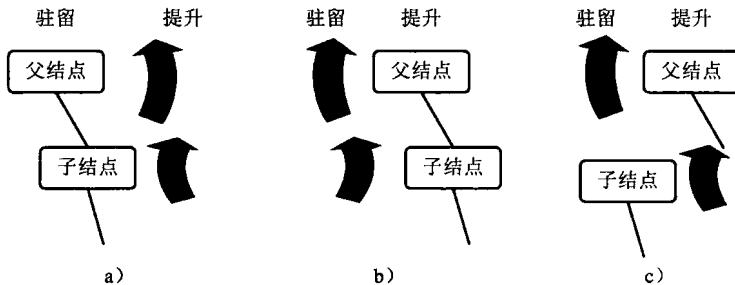


图 8-18 提升过程中父结点与子结点的交互。在图 a 中，父结点与子结点均不提升，被提升的结点（如果有的话）沿右边向上传递；在图 b 中，父结点与子结点同时提升，不提升的结点向上传递；在图 c 中，父结点被提升而子结点不提升，它们被分离，并与相同处置方式的其他结点一起向上传递。不可能将一个子结点从其父结点之下提升上来（参阅正文）

在消除公共子表达式和左移动提升中，对一个作为组成部分的变量的赋值将注销符号表中对应的入口。在右提升中，还须考虑对变量的引用：如果一个变量的赋值语句未被提升，则该变量的引用不可提升至该赋值语句之上；一个变量的赋值语句不可提升到该变量的引用之上，也不可提升到该变量的其他赋值语句之上。这一约束的实现利用了两个从左到右流动的集合：一个集合包含了仍有未提升引用的变量，另一集合包含了仍有未提升赋值语句的变量。仅当被赋值的变量同时不属于这两个集合时，该赋值语句才可提升；如果一个变量不属于仍有未提升赋值语句的变量集合，则该变量的引用可被提升。

8.8.6 无用代码以及其他从右到左的数据流分析

右移动提升本质上是模拟执行的从右到左镜像。消除无用代码也要求从右到左的分析以找出活跃变量。对死变量的赋值是无用的，因而可将其删除。本章之前已介绍了活跃变量分析。

有一种寄存器分配策略与活跃变量分析相关：首先寻找并确定一个值应存放在哪一个寄存器中，然后向后处理，将寄存器指派给子表达式，使得该值最终存放在该寄存器中。第 9 章讨论寄存器分配的常见课题时，将会更深入地探讨这一问题。高性能计算机中的资源调度也有类似的需求。

8.8.7 数学等式与代码选择

在大多数计算机中，不同指令的执行时间存在较大的差异；一个典型例子是乘法比加法或移位更慢。一个聪明的编译程序可利用诸如 $n + n = 2n$ 这类数学等式，用乘数自身的累加取代它与常量 2 的乘法，或令其向左移 1 位。与 2 的任何常量幂的乘法均可转换为适当位数的移位。这类优化称为强度削弱，因为实现某一计算的代码的强度（时间、空间）被削弱。

代码清单 8.12 中的代码生成文法片段展示了两种可应用到 Itty Bitty 栈机器的不同强度削弱方法，这些方法将与常量 2 相乘的乘法转换为加法（假设加法比乘法在性能上更有优势）。第一条规则将一个乘法结点变换为一个加法结点，方法是制作其子树的一个副本；这里假设接下来应用的消除公共子表达式优化将消除这一副本（并且在该子表达式中没有副作用）。另一条规则是代码生成（树平展）文法的一个片段，它基于不同的树形态生成不同的代码。当然，一个编译程序只会选用其中的一种方法。注意第一条规则在形式上是语义驱动的，而第二条规则是语法驱动的。利用条件属性断言和“最大咀嚼”树分析，在这两种情况下上述两种分析方法均可使用。

代码清单 8.12 强度削弱的两个文法片段

StrenRed	{ 在消除公共子表达式前执行 }
→ myself:<Star left rite> left:StrenRed rite:StrenRed ([rite:<CON>%2; xform = <Plus left left>] [left:<CON>%2; xform = <Plus rite rite>] [otherwise; xform = myself])	{ 检查子树 ... } { ... 是否符合强度削弱条件 } { 右子树是常量 2 } { 左子树是常量 2 } { 两棵子树均不符合条件 }
⇒ xform	{ 执行树的变换 }
CodeGen ↓inlocn:int ↑outlocn:int	{ 在最后的代码生成时执行 }
→ <Star left <CON>%2> left:CodeGen ↓inlocn ↑midlocn Emit ↓8 ↓midlocn ↑nxtlocn Emit ↓12 ↓nxtlocn ↑outlocn	{ 处理左子树 } { 输出 DUPE 操作码 } { 输出 ADD 操作码 }
→ <Star <CON>%2 rite> rite:CodeGen ↓inlocn ↑midlocn Emit ↓8 ↓midlocn ↑nxtlocn Emit ↓12 ↓nxtlocn ↑outlocn	{ 处理右子树 } { 输出 DUPE 操作码 } { 输出 ADD 操作码 }

→	<Star left rite>	
	[otherwise]	{ 上述两种情况之外的情况 }
	left:CodeGen ↓inlocn ↑midlocn	{ 处理左子树 }
	rite:CodeGen ↓midlocn ↑nxtlocn	{ 处理右子树 }
	Emit ↓l1 ↓nxtlocn ↑outlocn	{ 输出 MPY 操作码 }

常量折叠时还可挖掘另一些数学等式的巧妙用法。根据加法与乘法的分配律，涉及同一常量因子的两个乘积之和，可变换为一个和的乘积；而交换律和结合律在很多时候可用于收集常量项以实现部分求值，尽管这两个代数性质在计算机的算术运算中并非总是成立。Itty Bitty 栈机器中早已使用了关于负数的公理以执行减法；在寄存器机器上将负号分配到表达式中有时可节省一条中间的存储或装入指令。

强度削弱优化是更通用的一类优化技术“代码选择”的一部分。这类优化方法将分析目标硬件机器的不同操作码，为中间语言的每一结构在若干可能的选项中挑选最佳的指令序列。例如，在 Itty Bitty 栈机器上，逻辑非运算 **NOT** 将其操作数的所有位求反；而 Modula-2 语言的运算符 **NOT** 仅求反单个位以区别 **FALSE**（以 0 表示）和 **TRUE**（以 1 表示）。将 **FALSE** 的所有位求反结果为-1，这与 **TRUE** 不同。有多种 IBSM 指令序列的选择可实现一个正确的 **NOT** 运算，如表 8-4 所示；编译程序设计人员须根据每一指令序列的硬件执行时间和代码空间从中选择一项。

表 8-4 实现布尔运算符 NOT 的四种 IBSM 代码序列

NOT	ZERO	NEG	ONE
ONE	EQUAL	ONE	XOR
AND		ADD	

不同于 Itty Bitty 栈机器，许多计算机并没有特定的硬件将比较结果转换为单个位，使得这个位可存储到一个变量中，或用作布尔运算 **AND** 和 **OR** 的一个操作数。如果需要这一种转换，勤奋的编译程序设计人员将分析多种不同的指令序列，然后寻找一种高效的算法，正如之前我们对 IBSM 中布尔运算符 **NOT** 的处理。

8.8.8 循环结构分析

循环结构分析的大多数研究文献是关于如何从中间代码的一种线性表示（基本块或四元式）重构程序的结构图，对于 FORTRAN 这类非结构化语言则是别无选择。鉴于 Modula-2（以及在某种程度上 Pascal 和 C 语言，其中非结构化的语言特征要么被限制，要么往往被反对使用）这类结构化程序设计语言的广泛使用，我们更愿意在中间代码保留尽可能多的源语言结构，以便于循环分析。

在大多数现代语言中，任意循环的抽象树表示要求有两个不同的结点类型，如图 8-19 所示。整个循环由一个 **LOOP** 结点及其下的循环体表示，**LOOP** 结点下的所有东西均可能迭代，连续执行直至到达由一个 **EXIT** 结点表示的代码。如果由循环体子树表示的代码未遇到一个 **EXIT** 即结束执行，则循环从头开始重复执行。一些语言也可能在合适的地方显式地表示一个 **CYCLE** 结点；由于这不会另外带来新的困难或理解问题，我们将其细节留待读者练习。

循环分析的最重要结果是找出所谓的持久循环变量，即 PLV。持久循环变量是一个在循环体入口处活跃的变量，并且在循环体中被赋值。换言之，该变量先被引用，然后被修改。许多文献更关注归纳变量，该变量在每次循环迭代中以循环不变的量递增（或递减）。归纳变量通常可从一条 **FOR** 循环语句的控制变量导出，但也可在其他源代码的形式中显式地编码表示这些变量。控制变量本身也是一个持久循环变量；一些归纳变量优化方法将派生的归纳变量转换为独立的持久循环变量。

利用一次遍历循环体的数据流分析很容易找出持久循环变量，既可以从左到右，也可以从右到左。在数据流分析过程中必须传播两个集合，以正确地找出持久循环变量。其中一个是被赋值的变量的集合，另一是活跃的变量引用（从循环的开头看）的集合。在从左到右的数据流分析中，如果一个变量不在被赋值的变量集中，则将该变量添加到正在使用的活跃引用集中；在遇到一个变量的赋值语句时，该变量将无条件地添加到被赋值的变量集中。在从右到左的数据流分析方向中，仅当一个变量已在被赋值的变量集中，才将该变量添加到活跃引用集中；赋值语句则从活跃引用集中删除变量，同时更新被赋值的变量集。当数据流分析过程结束时，活跃变量集是循环体传播出来的两个正处理集合的交集。由于之前已深入讨论了数据流分析技术，我们将特定的文法细节留作练习。

在一次遍历循环体的（后续）模拟执行分析中，持久循环变量可用于找出循环不变量以实现代码移动。该功能有些类似常量折叠与代码提升的交叉，有时实际上又称为代码提升。如果一个变量既不是循环不变量也不是持久循环变量，则必须在循环体中使用它之前对它进行赋值。因而，可在模拟执行过程中向前传播一个集合，该集合包含了所有已知不是循环不变的变量；在循环体的开头，该集合初始化为持久循环变量集，并包含沿途被赋值的所有变量。如果一个表达式仅含常量和不属于该集合的变量引用，则可推断该表达式为循环不变的，并向前移出循环；如果所有实参均为循环不变的，则一个无副作用的函数也可看作是循环不变的。

代码清单 8.13 展示了一个循环不变代码移动的文法片段，该文法采用了图 8-8 所示的合成方法。这种方法并不是从嵌套循环向外沿途传播循环不变代码，尽管我们稍作修改即可如此。该文法中特别值得注意的是 **IF** 结点的控制表达式碰巧是循环不变的情况。该文法展示了该表达式将被移出循环，并赋值给一个新的临时变量；该变量在循环中被测试，以决定选择哪一个分支。

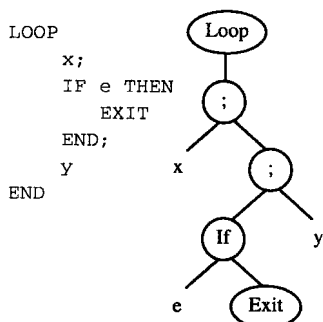


图 8-19 循环结构的结点，其中展示了显式的 EXIT 结点

代码清单 8.13 循环不变代码移动的文法片段

LoopCon ↓notcon:set ↑outnot:set ↑concode:tree

→	<Loop body>%plv	
	body:LoopCon ↓plv ↑newnot ↑moved	{ 处理循环体 }
	[concode = <>]	{ 不对外传播任何东西, }
	[union ↓notcon ↓newnot ↑outnot]	{ 除了被修改的变量集 }
⇒	<Semi moved <Loop body>%plv>	{ 将被移动代码加到循环前头 }
→	<Exit>	

```

[outnot = notcon; concode = <>]           { 结果根本没有任何东西 }

→ myself:<If expn then else>
  expn:ExpnCon ↓notcon ↑iscon ↑moved      { 处理表达式 }
  then:LoopCon ↓notcon ↑notl ↑movet       { 处理左子树 }
  else:LoopCon ↓notcon ↑notr ↑movee       { 处理右子树 }
  [union ↓notl ↓notr ↑outnot]             { 合并被修改的变量集 }
  ([iscon = false; xform = myself]        { 非循环不变, 合并所有待移动代码 }
  [concode = <Semi moved <Semi movet movee>>]
  |[otherwise; newvar ↑varno]              { 否则创建一个临时变量, }
  [xform = <If <VAR>%varno then else>]    { 以保存循环不变的表达式 }
  [concode = <Semi <Assn <VAR>%varno expn> <Semi movet movee>>]])
⇒ xform                                   { 若有需要则替换结点 }

→ <Semi left rite>
  left:LoopCon ↓notcon ↑midnot ↑movel     { 处理左子树 }
  rite:LoopCon ↓midcon ↑outnot ↑mover     { 处理右子树 }
  [concode = <Semi movel mover>]          { 合并待移动的代码 }

→ myself:<Assn <VAR>%varno expn>
  expn:ExpnCon ↓notcon ↑iscon ↑moved      { 处理表达式 }
  ([notinset ↓varno ↓notcon; iscon = true] { 表达式和变量为循环不变, }
  [concode = myself; outnot = notcon; xform = <>] { 故移动整条赋值语句 }
  |[otherwise; addset ↓varno ↓notcon ↑outnot] { 否则将变量添加到集合中 }
  [concode = moved; xform = myself])      { 并传递待移动的代码 }
⇒ xform                                   { 若有需要则替换结点 }

ExpnCon ↓notcon:set ↑iscon:bool ↑concode:tree

→ <CON>%val
  [concode = <>; iscon = true]             { 是的, 它是循环不变的 }

→ <VAR>%varno
  ([notinset ↓varno ↓notcon]               { 该变量是循环不变的 }
  [concode = <>; iscon = true]
  |[otherwise; concode = <>; iscon = false]) { 不是循环不变的 }

→ myself:<Plus left rite>                  { 对任意运算符结点, }
  [otherwise]                             { 上述情况之外 }
  left:ExpnCon ↓notcon ↑isconl ↑movel     { 处理左子树 }
  rite:ExpnCon ↓notcon ↑isconr ↑mover     { 处理右子树 }
  ([isconl = isconr; xform = myself; iscon = isconl] { 两个子表达式相同, }
  [concode = <Semi movel mover>]           { 故将结点放在一起 }
  |[isconl = true; iscon = false; newvar ↑varno] { 左为循环不变: 创建临时变量, }
  [xform = <Plus <VAR>%varno rite>]        { 以保存不变式的值 }
  [concode = <Semi <Assn <VAR>%varno left> mover>]
  |[isconr = true; iscon = false; newvar ↑varno] { 右边为循环不变而左边不是 }
  [xform = <Plus <VAR>%varno rite>]        { 处理方法相同 }
  [concode = <Semi <Assn <VAR>%varno left> mover>]])
⇒ xform                                   { 若有需要则替换结点 }

```

“循环开关外提”是一种优化技术，在上述情况下执行更为关键的处理，将一个循环变成两个循环，如图 8-20 所示。除分叉结点外，循环中的所有结点被复制。条件测试被移至循环之外，并据此选择两个副本中的其中一个；**THEN** 部分保留在循环的一个副本中，**ELSE** 部分则保留在另一副本中。因而，尽管代码的大小显然增大了，但所有的测试与分支逻辑均已从循环中删除，从而改进了执行时间，并有可能产生更多的循环不变优化和消除公共子表达式优化，而这些优化原本可能由于 **THEN** 与 **ELSE** 部分的冲突而无法实现。

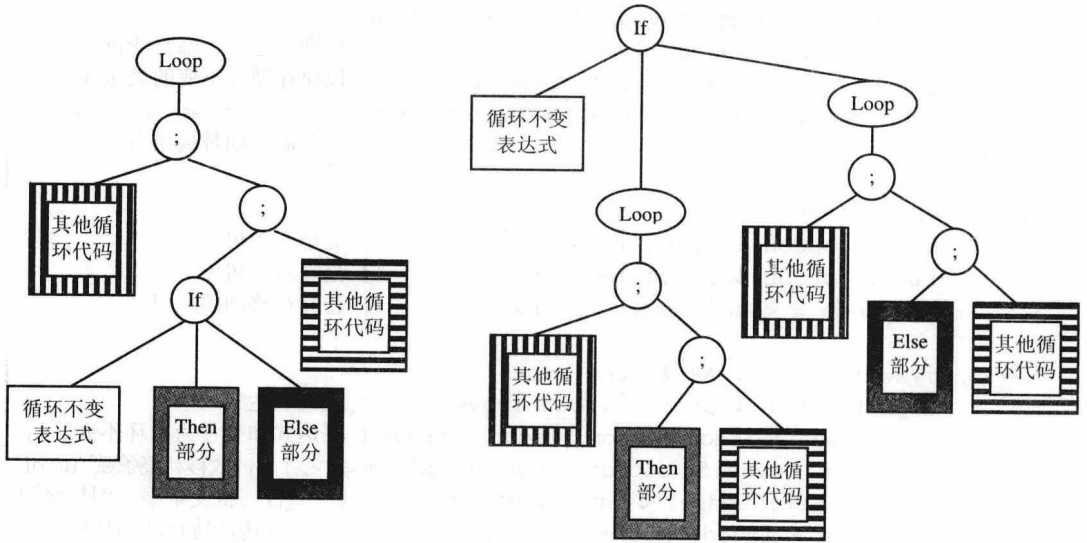


图 8-20 循环开关外提

8.9 实现抽象语法树

在已发表的研究文献中，很少涉及树变换程序的实现。本章余下内容的大多数材料描述了原创的工作，它们仍有很大的改进空间；此处的介绍主要是为了展示一种可能的实现方式，我们鼓励读者为自己的编译程序开发更高效的算法。

在内存中实现一个真正的抽象语法树，需使用指针和记录数据类型以实现类属的树结点。在一个手工编写的编译程序中，可有效地利用记录变体定义不同的结点结构，但通常更简单的方法是定义单个足够通用的结点记录类型，使得它可以处理编译程序中用到的所有不同的结点类型。例如，如果树结点最多有三棵子树，则结点记录应有三个字段存放子树的指针。使用统一的结点结构的另一好处是在变换过程中丢弃树的片段时，易于回收动态存储空间。与维护一个已分配的空结点列表的解决方案相比，由操作系统的 **DISPOSE** 过程完成这一功能显得有些低效。类似地，当内存的约束强制规定使用虚拟内存实现技术时，统一结点的大小将简化这一任务。

类属结点记录结构的最小需求是：有一个结点名定义了该结点的类别是什么，每一可能的子树有一个指针，以及某种类型的一个指针以容纳一个装饰。在装饰字段中或整个结点记录中的变体记录支持多态的装饰：通常一棵抽象语法树的不同部分（即不同类别的结点）会规定有不同类型的装饰。

代码清单 8.14 展示了一类结点 (**WHILE**) 以集合为装饰，而另一类结点 (**Assign**、**ID** 和

NUM) 以整数为装饰。一种实现方式还可能在这些记录变体中包含标签字段, 并且在没有强类型的编译程序构造工具可用时, 还可包含一些代码检查其用法的一致性; 在编译程序通过彻底的测试后, 投入生产用的编译程序若有需要改进编译的时间效率, 可删除这些一致性检查代码和标签字段。

代码清单 8.14 一个树结点的实现模块

```

MODULE Trees;
EXPORT
  Tree, Sett, NodeKind, Nill, (* 基本类型 *)
  BuildTree, ParseTree, DisposeTree, TransformTree, (* 过程 *)
  IntTree, TreeInt, IntSett, (* 整数的强制类型转换 *)
  Union, Intersect, SubSett, InSett; (* 集合运算过程 *)

CONST
  Nill = NIL;
  SegSize = 64; (* 集合段的大小, 根据硬件选择 *)

TYPE
  Tree = POINTER TO Node;
  Sett = Tree;
  NodeKind = (None, Int, Setn, IDn, NUMn, Assign, Readn, Writen, IFn, WHILEn, Semi);
  (* 有需要还可添加其他结点类型 *)

  BitVector = SET OF [0 .. SegSize - 1];
  Node = RECORD
    CASE NodeName: NodeKind OF
      Int:
        IntValue: INTEGER; |
      Setn:
        BaseValue: INTEGER;
        Link: Sett;
        Segment: BitVector; |
      ELSE
        Left, Middle, Right, Decor: Tree;
    END
  END;

PROCEDURE Union(a, b: Sett): Sett;
VAR
  theTree, theRest: Sett;
BEGIN
  IF (a = b) OR (b = NIL) THEN
    RETURN a
  ELSIF a = NIL THEN
    RETURN b
  ELSE
    theRest := Union(a^.Link, b^.Link);
    IF (theRest = a^.Link) AND (b^.Segment <= a^.Segment) THEN
      RETURN a
    ELSIF (theRest = b^.Link) AND (a^.Segment <= b^.Segment) THEN
      RETURN b
    ELSE

```

```

    NEW(theTree);
    WITH theTree^ DO
        NodeName := Setn;
        BaseValue := a^.BaseValue;
        Segment := a^.Segment + b^.Segment;
        Link := theRest
    END;
    RETURN theTree
END
END
END Union;

PROCEDURE Intersect(a, b: Sett): Sett;
VAR
    theTree, theRest: Sett;
BEGIN
    IF a = b THEN
        RETURN a
    ELSIF (a = NIL) OR (b = NIL) THEN
        RETURN NIL
    ELSIF a^.BaseValue > b^.BaseValue THEN
        RETURN Intersect(a, b^.Link)
    ELSIF a^.BaseValue < b^.BaseValue THEN
        RETURN Intersect(a^.Link, b)
    ELSIF a^.Segment * b^.Segment = BitVector{} THEN
        RETURN Intersect(a^.Link, b^.Link)
    ELSE
        theRest := Intersect(a^.Link, b^.Link);
        IF (theRest = a^.Link) AND (a^.Segment <= b^.Segment) THEN
            RETURN a
        ELSIF (theRest = b^.Link) AND (b^.Segment <= a^.Segment) THEN
            RETURN b
        ELSE
            NEW(theTree);
            WITH theTree^ DO
                NodeName := Setn;
                BaseValue := a^.BaseValue;
                Segment := a^.Segment * b^.Segment;
                Link := theRest
            END;
            RETURN theTree
        END
    END
END Intersect;

PROCEDURE InSett(theTree: Sett; Value: INTEGER): BOOLEAN;
BEGIN
    IF theTree = NIL THEN
        RETURN FALSE
    ELSE
        WITH theTree^ DO
            IF BaseValue > Value THEN
                RETURN FALSE
            ELSIF BaseValue + SegSize <= Value THEN
                RETURN InSett(Link, Value)
            END IF
        END
    END
END InSett;

```

```

    ELSE
        RETURN (Value MOD SegSize) IN Segment
    END
END
END
END InSett;

PROCEDURE DisposeTree(theTree: Tree);
BEGIN
    (* 应递归地处置 theTree 及其所有子树 *)
END DisposeTree;

PROCEDURE SubSett(a, b: Sett): BOOLEAN; (* 当且仅当 a <= b 时返回真 *)
BEGIN
    IF a = NIL THEN
        RETURN TRUE
    ELSIF b = NIL THEN
        RETURN FALSE
    ELSIF a^.BaseValue > b^.BaseValue THEN
        RETURN SubSett(a, b^.Link)
    ELSIF a^.BaseValue < b^.BaseValue THEN
        RETURN (a^.Segment = BitVector{}) AND SubSett(a^.Link, b)
    ELSE
        RETURN (a^.Segment <= b^.Segment) AND SubSett(a^.Link, b^.Link)
    END
END SubSett;

PROCEDURE BuildTree(name: NodeKind; first, second, third, decn: Tree): Tree;
VAR
    theTree: Tree;
BEGIN
    NEW(theTree);
    WITH theTree^ DO
        NodeName := name;
        Left := first;
        Middle := second;
        Right := third;
        Decor := decn
    END;
    RETURN theTree
END BuildTree;

PROCEDURE ParseTree(theTree: Tree; name: NodeKind
    ; VAR first, second, third, decn: Tree): BOOLEAN;
BEGIN
    IF theTree = NIL THEN
        RETURN name = None
    ELSE
        WITH theTree^ DO
            IF NodeName <> name THEN
                RETURN FALSE
            ELSE
                first := Left;
                second := Middle;
                third := Right;
                decn := Decor;
            END
        END
    END
END ParseTree;

```

```

        RETURN TRUE
    END
END
END
END ParseTree;

PROCEDURE TransformTree(theTree, newTree: Tree);
BEGIN
    IF theTree <> NIL THEN
        IF newTree = NIL THEN
            WITH theTree^ DO
                NodeName := None;
                Left := NIL;
                Middle := NIL;
                Right := NIL;
                Decor := NIL
            END
        ELSE
            theTree^ := newTree^
        END
    END
END TransformTree;

PROCEDURE TreeInt(theTree: Tree): INTEGER;
BEGIN
    IF theTree = NIL THEN
        RETURN 0
    ELSE
        RETURN theTree^.IntValue
    END
END TreeInt;

PROCEDURE IntTree(Value: INTEGER): Tree;
VAR
    theTree: Tree;
BEGIN
    NEW(theTree);
    theTree^.NodeName := Int;
    theTree^.IntValue := Value;
    RETURN theTree
END IntTree;

PROCEDURE IntSett(Value: INTEGER): Sett;
VAR
    theTree: Sett;
BEGIN
    NEW(theTree);
    theTree^.NodeName := Setn;
    theTree^.BaseValue := Value - Value MOD SegSize;
    theTree^.Segment := BitVector(Value MOD SegSize);
    theTree^.Link := NIL;
    RETURN theTree
END IntSett;

END Trees;
```

当一个模块导出树结点的构建和分析过程时，还应同时导出树结点的类型。如果将一个集合（即集合的位向量表示）实现为树结点类型的变体，该模块还须导出操纵这些集合的过程。代码清单 8.14 展示了这一模块的一种简单实现，但其中未提供存储空间的恢复功能。尽管这里展示的是一个内部模块，但在实际应用中它显然可划分为两个分别编译的定义模块和实现模块。

树结点的构造过程 **BuildTree** 相当简单，它返回一个指向树的指针，可直接用作另一次 **BuildTree** 调用的参数，从而支持将复杂的树模板直接翻译为可执行的代码。对树结点的分析稍复杂一些。每一个单独的 **ParseTree** 函数调用识别一个结点，并以变参形式返回一棵子树。借助于嵌套的 **IF** 结构可分析复杂的模板。代码清单 8.16 中的样板变换程序演示了这两个过程。过程 **TransformTree** 用于以某一其他结构取代一个结点结构，它无法将一棵空树变换为一个非空的树结点，反之亦然；但这些限制可以机械地强制执行。

不同于 **Modula-2** 和 **Pascal** 等语言中的大多数集合类型，用于数据流分析的位向量本质上具有无限数目的元素。编译一个带有大量变量声明的程序时，分析过程需要相应的大型位向量。而同一个编译程序运行在一个较小内存分区中编译小型程序时，不应承担因大型程序对大型集合的需求而带来的（时间或空间）开销。因而，这类位向量的最实用实现方案是一个动态链接的集合段链表。代码清单 8.14 中的实现演示了如何完成这一方案。尽管其中的集合操纵过程表示为递归形式，但由于许多硬件和软件环境限制带来的过程调用开销，采用迭代的实现方式可能会更加高效。某些编译程序假设一个合理的过程中变量数目不会超过某一上限，简单地给变量数目设置一个武断的上限（譬如 256 个）；这要求局部变量总是从 0 开始编号，非局部变量（如果数据流分析算法完全支持非局部变量）则重新映射为该上限之内的引用编号，只要仍有未使用的位向量赋值。

代码清单 8.14 的实现在设计时通过禁止空集合段，将一个稀疏集合中链接在一起的集合段的数目减少到最小数目。单元素集中的元素仅使用一个段结点，并且交运算将删除作为结果的空段。并运算和交运算均尽可能地尝试复用集合段。一个段中的元素数目应根据硬件作相应调整，尽可能多地用一次原子的机器操作访问一个段；这一目标还须同时平衡与由子树指针的数目所决定的结点大小之间的关系，将内存浪费减少至最小。编译一个真正有意义的程序涉及内存中树表示的大规模内存用法，因而这是一个重要的考虑因素。

如果要求一个树变换程序处理超出常驻内存所能容纳的结点，树结点的实现很容易转换为一个虚拟内存方案，除了一个随机访问文件之外，无要求有操作系统的支持。我们可以在一个结点块的文件中显式地分配结点的空间，而不是使用 **NEW** 和 **DISPOSE** 命令管理指针。**Tree** 类型不再是一个指针，而是一个指向文件的基数索引。**Modula-2** 这类现代程序设计语言的信息隐藏能力的优点之一，是树变换程序的客户例程无须修改，即可适应树结点实现的这种彻底改变。代码清单 8.15 展示了在一个虚拟内存版本的树结点模块中，文件管理方面是如何实现的。

代码清单 8.15 一个基于虚拟内存的树结点模块

```
MODULE Trees;
FROM FileSystem IMPORT (* 或等效的其他模块 *)
    BinaryFile, (* 不透明的文件类型 *)
    RandomOpen,
    (* filevar: BinaryFile; maxBytes: CARDINAL *)
```

```

RandomRead,
  (* filevar: BinaryFile; position: CARDINAL; block: ADDRESS; nBytes: CARDINAL *)
RandomWrite;
  (* filevar: BinaryFile; position: CARDINAL; block: ADDRESS; nBytes: CARDINAL *)

EXPORT
  Tree, NodeKind, Nill,                (* 基本类型 *)
  BuildTree, ParseTree, DisposeTree,    (* 过程 *)
  IntTree, TreeInt, IntSett,            (* 整数的强制类型转换 *)
  Union, Intersect, SubSett, InSett;    (* 集合运算过程 *)

CONST
  Nill = 0;

CONST
  BlockSize = 1024;                    (* 一个块中的结点数 *)
  MemBlocks = 50;                      (* 内存中块的数目 *)
  MaxBlocks = 1000;                   (* 文件中块的最大数目 *)

TYPE
  Tree = CARDINAL;
  ...
  NodeBlock = ARRAY [0 .. BlockSize - 1] OF Node;

CONST
  BlockBytes = SIZE(NodeBlock);        (* 一个块占用的字节数 *)

VAR
  NodeFile: BinaryFile;                (* 磁盘文件, 用于存放被交换出的树 *)
  LastTree: Tree;                      (* 已分配的树的数目 *)
  NextTime: CARDINAL;                 (* 当前的时间 *)
  LastBlock: CARDINAL;                (* 文件扩展 *)

  InMemoryTrees: ARRAY [1 .. MemBlocks] OF NodeBlock;
  InMemoryInfo: ARRAY [1 .. MemBlocks] OF RECORD
    Blockno: [0 .. MaxBlocks - 1];
    Dirty: BOOLEAN;                  (* 为真则将此块写入磁盘 *)
    Age: CARDINAL;                  (* 用于计算最近最少使用 (LRU), 越小的数表示用得越久 *)
  END;
  BlockIndex: ARRAY [0 .. MaxBlocks - 1] OF [0 .. MemBlocks];
  (* 指向 InMemoryTrees 的索引, 否则为 0 *)

PROCEDURE Touch(theTree: Tree; MakeDirty: BOOLEAN): CARDINAL;
  (* 返回指向 InMemoryTrees 的索引 *)

VAR
  lru, i, k: CARDINAL;
BEGIN
  i := BolckIndex[theTree];
  IF i = 0 THEN                      (* 不在内存中, 找最老的块交换 *)
    lru := NextTime;
    FOR k := 1 TO MemBlocks DO
      WITH InMemoryInfo[k] DO
        IF Age < lru THEN
          i := k;
          lru := Age
        END IF;
      END WITH;
    END FOR;
  END IF;
  BlockIndex[i] := theTree;
  IF MakeDirty THEN
    NodeFile.Append(NodeFile.Position, NodeBlock[i]);
    NodeFile.Position := NodeFile.Position + BlockBytes;
  END IF;
  i := 0;
END Touch;

```

```

        END
    END
    END; (* FOR k *)
    WITH InMemoryInfo[i] DO
        IF Dirty THEN
            RandomWrite(NodeFile, Blockno * BlockBytes, InMemoryTrees[i], BlockBytes)
        END;
        Blockno := theTree DIV BlockSize;
        IF Blockno > LastBlock THEN
            RandomWrite(NodeFile, Blockno * BlockBytes, InMemoryTrees[i], BlockBytes);
            LastBlock := Blockno
        ELSE
            RandomRead(NodeFile, Blockno * BlockBytes, InMemoryTrees[i], BlockBytes)
        END;
        Age := NextTime;
        Dirty := MakeDirty
    END (* WITH InMemoryInfo[i] *)
ELSE
    WITH InMemoryInfo[i] DO
        Age := NextTime;
        IF MakeDirty THEN
            Dirty := TRUE
        END
    END (* WITH InMemoryInfo[i] *)
END; (* IF i = 0 *)
NextTime := NextTime + 1;
RETURN i
END Touch;

PROCEDURE NewTree(): Tree;
BEGIN
    LastTree := LastTree + 1;
    RETURN LastTree
END NewTree;

PROCEDURE BuildTree(name: NodeKind; first, second, third, decn: Tree): Tree;
VAR
    theTree: Tree;
BEGIN
    theTree := NewTree();
    WITH InMemoryTrees[Touch(theTree, TRUE)][theTree MOD BlockSize] DO
        NodeName := name;
        Left := first;
        Middle := second;
        Right := third;
        Decor := decn
    END;
    RETURN theTree
END BuildTree;

PROCEDURE ParseTree(theTree: Tree; name: NodeKind
    ; VAR first, second, third, decn: Tree): BOOLEAN;
BEGIN
    IF theTree = Nil THEN
        RETURN name = None
    ELSE

```

```

        WITH InMemoryTrees[Touch(theTree, FALSE)][theTree MOD BlockSize] DO
            IF NodeName <> name THEN
                RETURN FALSE
            ELSE
                first := Left;
                ...
        END ParseTree;
    ...

PROCEDURE InitTrees;                                (* 模块初始化代码 *)
VAR
    n: CARDINAL;
BEGIN
    RandomOpen(NodeFile, MaxBlocks * BlockBytes);    (* 或诸如此类的代码 *)
    LastTree := Nil;
    NextTime := 1;
    FOR n := 1 TO MaxBlocks DO
        IF n < MemBlocks THEN
            BlockIndex[n - 1] := n
        ELSE
            BlockIndex[n - 1] := 0
        END
    END;
    FOR n := 1 TO MemBlocks DO
        WITH InMemoryInfo[n] DO
            Age := 0;
            Blockno := n - 1;
            Dirty := FALSE;
            RandomWrite(NodeFile, Blockno * BlockBytes, InMemoryTrees[n], BlockBytes)
        END
    END;
    LastBlock := MemBlocks - 1
END InitTrees;

BEGIN
    InitTrees;
END Trees;

```

在这一实现中，函数 **Touch** 完成了所有的重要工作。给定一个虚拟的树指针（即一个指向文件的索引），该函数判定该块是否在内存中；如果不在，则找出一个最近最少使用的块与该块交换。该函数返回一个指向内存数组的索引，指明了包含该结点的块。以类似方式索引的另一信息数组保存了最近访问时间的记录（依据一个每次调用时累加的计数器），以及一个表示该块被置换时是否需要写入磁盘的标志。对于特殊的引用分布情况还存在更好的交换算法，但上述算法对于大多数情况已经是相当好了。

8.10 实现 TAG 驱动(tree)的树变换

给定如代码清单 8.14 所示的树结点实现，根据一个 TAG 构造一个树变换程序又再成为机械抄写的体力活。我们从第 4 章介绍的递归下降构造方法出发，为它扩展如第 5 章所示的属性求值。对树模板的分析是通过适当地调用导入的函数 **ParseTree** 来完成的；轮流尝试每一模板，从每一个失败的产生式回退，直至所有断言求值为真。因而，一个非终结符的最简单实现方式是一个布尔函数，如果没有产生式可分析至结束，则该函数返回假。在某些情况下，数据

流分析可证明分析过程是确定的，并且可消除条件测试。表 8-5 给出了将每一种文法形式机械翻译为 Modula-2 语言的方案。代码清单 8.16 具体演示了代码清单 8.13 的循环不变 TAG 中前三条产生式的翻译代码。

表 8-5 将 TAG 形式翻译为 Modula-2 语言的代码

文法形式	Modula-2 代码
Nonterminal... → ...;	PROCEDURE NonTerminal(theTree: Tree; ...): BOOLEAN; VAR ok: BOOLEAN; any: Tree; BEGIN ok := TRUE; ... RETURN ok END NonTerminal;
↓inherited: typename	inherited: typename;
↑derived: typename	VAR derived: typename;
→ <Kind left rite>%deco ...	IF NOT ok THEN ok := ParseTree(theTree, Kind, left, rite, any, deco); IF ok THEN ... END END;
→ atree: <Top <Inner>> ...	IF NOT ok THEN atree := theTree; ok := ParseTree(atree, Top, temp, any, any, any); IF ok THEN ok := ParseTree(temp, Inner, any, any, any, any); ... END END;
atree: NonTerm ↓inhatt ↑synatt	IF ok THEN ok := NonTerm(atree, inhatt, synatt) END;
{[attr = value] ... [otherwise] ...}	IF ok THEN IF attr = value THEN ... ELSE ... END END;
{attr = <Kind left rite>%deco}	IF ok THEN attr := BuildTree(Kind, left, rite, nil, deco) END;
⇒ atree	IF ok THEN TransformTree(theTree, atree) END;

代码清单 8.16 根据代码清单 8.13 中 TAG 编写的一个样板树变换程序

```

PROCEDURE LoopCon(theTree: Tree; notcon: Sett
    ; VAR outnot: Sett; VAR concode: Tree): BOOLEAN;

VAR
    ok: BOOLEAN;
    body, moved, expn, thenn, ellse, movet, movee, xform, temp: Tree;
    plv, newnot, notl, notr: Sett;

```

```

iscon: BOOLEAN;
varno: INTEGER;

BEGIN
  ok := ParseTree(theTree, Loop, body, temp, temp, plv);
  IF ok THEN
    ok := LoopCon(body, plv, newnot, moved)
  END;
  IF ok THEN
    concode := Nill;
    outnot := Union(notcon, newnot);
    TransformTree(theTree, BuildTree(Semi, moved, BuildTree
      (Loop, body, Nill, Nill, plv)
      , Nill, Nill));
  END;
  IF NOT ok THEN
    ok := ParseTree(theTree, Exit, temp, temp, temp, temp)
  END;
  IF ok THEN
    outnot := notcon;
    concode := Nill
  END;
  IF NOT ok THEN
    ok := ParseTree(theTree, Iff, expn, thenn, ellse, temp)
  END;
  IF ok THEN
    ok := ExpnCon(expn, notcon, iscon, moved)
  END;
  IF ok THEN
    ok := LoopCon(thenn, notcon, notl, movet)
  END;
  IF ok THEN
    ok := LoopCon(ellse, notcon, notr, movee)
  END;
  IF ok THEN
    outnot := Union(notl, notr)
  END;
  IF ok THEN
    IF iscon = FALSE THEN
      xform := theTree;
      concode := BuildTree(Semi, moved, BuildTree(Semi, movet, movee, Nill, Nill)
        , Nill, Nill)
    ELSE
      newvar(varno);
      xform := BuildTree(Iff, BuildTree(Varr, Nill, Nill, Nill, IntTree(varno))
        , thenn, ellse, Nill);
      concode := BuildTree(Semi, BuildTree(Assign, BuildTree(Varr, Nill, Nill, Nill
        , IntTree(varno)), expn, Nill, Nill), BuildTree(Semi, movet, movee, Nill, Nill)
        , Nill, Nill);
    END
  END;
  END;
  IF ok THEN
    TransformTree(theTree, xform)
  END;
  RETURN ok
END LoopCon;

```

代码清单 8.14 中的 TAG 实现以及表 8-5 存在一种危险，必须通过以上所示代码之外的其他方式来预防。问题出在抽象语法树的一个特定结点被变换时。由于树的其他部分或其他活动着的非终结符（活跃过程）可能引用了正被变换的结点，过程 **TransformTree** 必须用置换结点中内容的副本来替换结点记录中的内容；如果用于置换的树包含原结点作为一棵子树，变换结果将导致图 8-21 所示的循环结构，而不是最终想要的树。在循环不变代码的移动和提升优化中很可能出现这种情况，其中分号结点构建在引用结点之上。在手工实现变换文法时，编译程序设计人员须预防这一危害。诸如 TAG 编译程序这类机械的“编译程序—编译程序”则可通过适当的数据流分析检测这一错误，并制作一个引用结点的新副本加以改正。代码清单 8.16 中的样板代码通过重构引用结点正确地避免了这一问题（请注意图 8-21 所示两个文法之间的区别）。

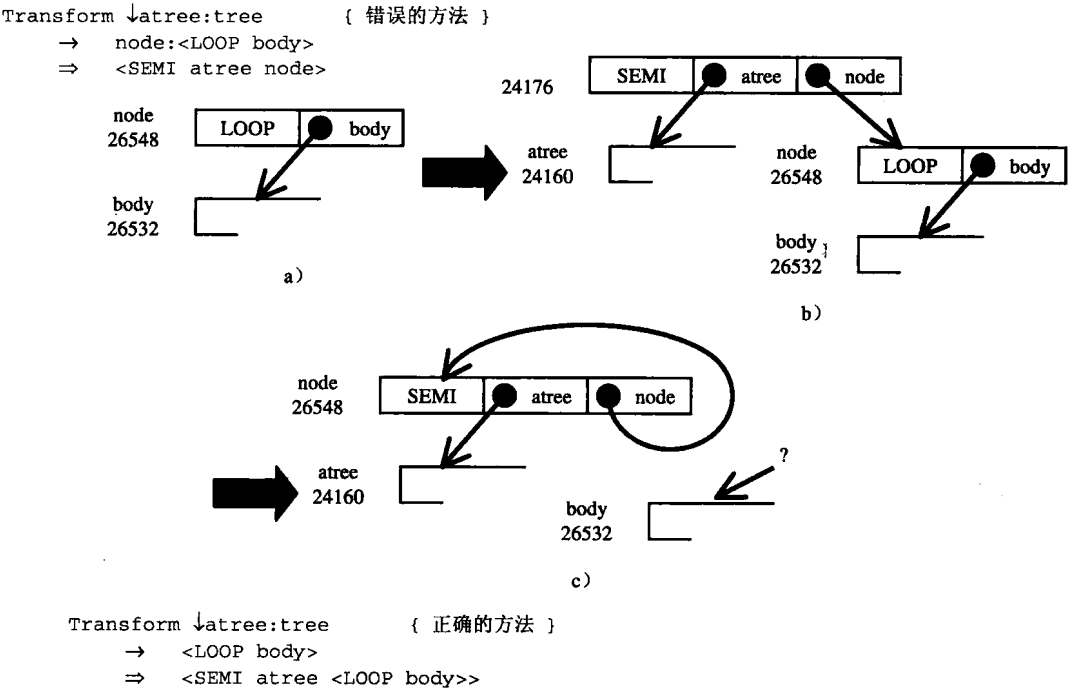


图 8-21 树变换的错误方法。第一步（图 b）在现有 LOOP 结点（图 a）之上构建了一个新的 SEMI 结点。最后一步（图 c）将新的 SEMI 结点复制到原树之上，导致一个循环链接。图中的数字表示物理内存位置。第二个文法则会构建一个新的 LOOP 结点，从而当原树被新树取代时，与循环体的连接将保持如图 b 所示的情况

小结

本章介绍了变换属性文法（TAG），它提供了一种语言用于定义优化变换和代码生成。一个 TAG 既是一个树变换文法，同时也是一个属性文法。树变换文法（TTG）定义了一个从输入文法的树到输出文法的树之间的映射。回顾第 5 章，属性文法（AG）是一个上下文无关文法，并为其中的每一非终结符扩展了相关联的属性。属性的求值与产生式的应用联系在一起。属性值可在分析树中向上或向下传递。TAG 的定义域是中间低级树（ILT）的集合，即编译程序前端所产生程序的树结构的中间变换形式。ILT 与分析树的区别在于它尽可能多地包含了低级的类机器代码细节，并且可删除无语义动作的非终结符。

本章介绍了树文法的表示法，它不受输入单词固有次序的约束，而是由编译程序设计人员负责将适

当的属性求值次序告知编译程序生成工具，这就有可能避免对属性文法的大量分析以确定一种高效的或最优的求值次序。

本章还介绍了如何将中间代码树构造为串文法中的综合属性，并作为一种有效途径连接编译程序前端的串文法与后端定义优化和代码生成的树文法。本章描述了代码优化变换的数据流分析用法，并给出了实用变换优化的综述。

符号

- ⇒ 分隔一个 TAG 产生式的右部。
- ∩ 集合的交。
- ∪ 集合的并。

缩略词

- AST Abstract Syntax Tree, 抽象语法树。
- CSE Common Subexpression Elimination, 消除公共子表达式。
- DFA Data-Flow Analysis, 数据流分析。
- ILT Intermediate Low-level Tree, 中间低级树。
- PLV Persistent Loop Variables, 持久循环变量。
- TAG Transformational Attribute Grammar, 变换属性文法。
- TTG Tree-Transformational Grammar, 树变换文法。

关键术语

abstract syntax tree (抽象语法树)

- (1) 从分析树中删除不必要的信息后，得到源程序的更高效表示。
- (2) 提炼了源程序语法结构的分析树（不保留空格、换行符、标识符和关键字的拼写等）。

data-flow algorithm (数据流算法)

iterated, monotonic (迭代的、单调的) 在每一次迭代中，集合始终如一地递增或递减，且不可改变增减的方向。

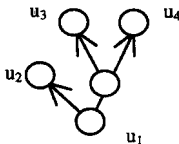
DFA (数据流分析) 数据流分析用于在整个计算单元中追踪数据的流向。

decoration (装饰) 与一棵抽象语法树中某一结点建立适当关联的任意数据。

generator template (生成程序模板) 定义了输出语言。

graph (图) 图 $G(V, E)$ 是一个如下的结构：它由顶点集 $V = \{v_1, \dots\}$ 、边集 $E = \{e_1, \dots\}$ 以及关联函数 $f: E \rightarrow V \times V$ 组成；例如在 $f(e_i) = (v_j, v_k)$ 中， e_i 属于 E ， v_j 和 v_k 属于 V 。二元组 (v_j, v_k) 标明了一条从 v_j 连接到 v_k 的边 e_i 的端点。

directed graph (有向图) 一个图，其中每一条边的端点都是有序的。在以下所示的图中，边 (u_1, u_2) 在图中，但 (u_2, u_1) 不在图中，因为不存在一条从 u_2 连接到 u_1 的边。



lattice (格) 集合 L 以及交运算 “ \cap ” 和并运算 “ \cup ” 称为格，当且仅当 L 是一个偏序集，并且对 L 中的任意 x 、 y 和 z 以下公理成立：

(1) $x \cap y = y \cap x$ 且 $x \cup y = y \cup x$ (交换律)

(2) $(x \cap y) \cap z = x \cap (y \cap z)$ 且
 $(x \cup y) \cup z = x \cup (y \cup z)$ (结合律)

(3) $x \cup (x \cap y) = x$ 且
 $x \cap (x \cup y) = x$ (吸收律)

match template (匹配模板) 定义了一个变换文法中的输入语言。

partial order (偏序) 集合 X 上的二元关系 \leq 是 X 的一个偏序, 如果它是自反的、传递的和反对称的, 即对 X 中的任意 x 、 y 和 z 均满足:

R: $x \leq x$ (自反性)

T: $x \leq y$ 且 $y \leq z$ 则蕴含 $x \leq z$ (传递性)

A: 存在 x 、 y , 使得 $x \leq y$ 成立, 但 $y \leq x$ 不成立 (反对称性)

例子: 自然数集合上的 \leq 关系; 一个集合的幂集上的 \supseteq 关系。

transformational grammar (变换文法) 一个上下文无关文法, 其产生式形如 “ $P \rightarrow \langle \text{匹配模板} \rangle \Rightarrow \langle \text{生成程序模板} \rangle$ ”。

transformations (变换)

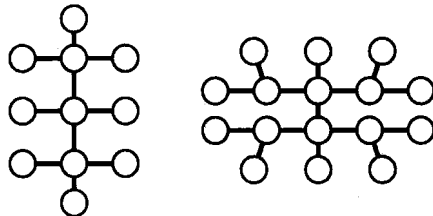
code elimination (代码消除) 从程序树中剪枝, 丢弃多余的分支。

code motion (代码移动) 将代码从程序中频繁执行的地方移至不经常使用的地方。

code selection (代码选择) 对树进行装饰, 以备在最后一遍中将树平展以生成代码。

translation grammar (翻译文法) 同时产生输入语言和输出语言。

tree (树) 如果一个连通、非空的图不含封闭路径 (即无环的), 则称其为树。以下为树的两个例子:



注意: 一个有向图是一棵有向树, 如果它有一个根结点, 从该结点出发到每一其他结点均存在一条有向路径, 并且作为其基础的无向图是一棵树。

tree grammar (树文法) 是一个文法, 其输入字母表由树结点及其连接组成, 而不是由字符或从一个线性串扫描得到的单词组成。

练习

1. 使用本章介绍的前缀波兰表示法, 将以下程序片段改写为树。标出运算符结点的名字, 从而 $a + 2$ 将写为 $\langle \text{Plus} \langle \text{ID} \rangle \% a \langle \text{NUM} \rangle \% 2 \rangle$ 。

(a) $(x + y) / (x - y)$

(b) $\text{next} := \text{temp} - \text{base} * 3 + 1$

(c) $a := 1; b := -a$

(d) IF $\text{tx} < \text{ty} + \text{ent}$ THEN $\text{ab} := -1$ ELSE $\text{ab} := 0$ END

(e) REPEAT

$\text{sum} := \text{sum} + v * v;$

IF $(v > \text{eps})$ AND $(v < \text{ovf})$ THEN $v := v + \text{delta}$ END

UNTIL $\text{sum} \geq \text{goal}$

(f) $\text{struct.lnk}^{\wedge} \text{.fld}$

2. 将代码清单 8.1 中的文法应用到以下表达式树, 尽可能多地折叠常量。给出所应用的产生式, 包括属

性流。假设符号表包含以下这些标识符：

dan	TRUE, 6
paul	FALSE, 88
sam	TRUE, 10

- (a) `<Minus <ID>%paul <Star <ID>%dan <NUM>%4>>`
- (b) `<Plus <Star <ID>%dan <NUM>%3> <Divd <ID>%sam <NUM>%2>>`
- (c) `<Plus <Plus <ID>%paul <NUM>%5> <ID>%sam>`

- 将练习 1 中的程序片段改写为四元式，并画出所有的基本块。
- 重新考虑代码清单 8.4 中的向前数据流问题，但选用的集合应保证你的算法能够捕获对未定义变量（例如变量 c）的使用。
- 修改代码清单 8.3 中的程序，使之包含一条对变量 c 进行赋值的语句，然后重新考虑代码清单 8.4 和 8.5 中的向前和向后数据流分析问题，并证明这一修改的正确性。可使用练习 4 中改进的向前数据流算法。
- 修改代码清单 8.6 的数据流分析属性文法，使之能找出单个循环中的持久循环变量。

复习小测验

指出下列陈述是否正确。

- 抽象语法树是分析树的另一种形式。
- 树文法的输入字母表由从线性串中扫描得到的单词组成。
- TAG 生成的树与程序树具有相同的根结点。
- 在一个变换文法中，输入语言和输出语言本质上是不同的语言。
- 一个 TAG 是一个 TTG，因而也是一个 AG。
- 文法不会受限于输入文件中任何固有的单词次序。
- 借助于一个识别串文法并且以一个树文法作为其翻译部分的翻译文法，可建立编译程序前端的串文法与定义了优化与代码生成的树文法之间的联系。
- 装饰是与一个抽象语法树的结点相关联的任意数据集。
- 与串文法相比，树文法完全是确定的。
- 最大咀嚼启发式方法用于在变换文法中将结点与模板匹配时，选用最小的、最精确的产生式模板。

编译程序实验项目

- 改写你的编译程序文法，将其前端和后端分离，从而由前端构建一棵树，后端根据该树生成 ISBM 代码。
- 选择本章讨论的一种或多种优化方法，为每一种优化编写一个完整的 TAG。
- 将你的文法合并为一个 TAG，然后在 TAG 编译程序上编译它；或根据你的文法，使用 Modula-2 之类的程序设计语言编写一个优化编译程序。

进一步阅读

Aho, A.V. & Ullman, J.D. *The Theory of Parsing, Translation, and Compiling: Vol. 2, Compiling*. Englewood Cliffs, NJ: Prentice Hall, 1973.

介绍了数据流分析中可用表达式的表示法。如果表达式 $A + B$ 的计算总是在到达某一块之前、但不是在 A 或 B 的定值之前，则称 $A + B$ 在该块入口处是可用表达式。

Backus, J. "The History of FORTRAN I, II, and III." *ACM SIGPLAN Proceedings of History of Programming Languages* (1981), pp.25-74.

Cocke, J. & Schwartz, J.T. *Programming Languages and Their Compilers. Preliminary Notes*, (2nd rev. version).

New York: Courant Institute of Mathematical Science, 1970.

在代码优化中使用了图抽象技术（参阅以下 Kuhn 等人讨论图抽象的文献注释）。

Deransart, P., Jourdan, M., & Lorbo, B. *A Survey on Attribute Grammars in Three Parts*. TR 485, Institut National de Recherche en Informatique, 1986.

第 I 部分“属性文法的主要成果”，参阅第 4 小节“基于树遍历的求值”。第 II 部分“现有系统综述”，参阅第 79ff 页的 MUG2，综述了一个基于属性文法的完整编译程序生成工具，该属性文法使用带属性的树变换作为工具，定义并执行源代码级的优化。第 III 部分“分类参考文献”，参阅第 41ff 页的数据流分析与代码生成。

Engelfriet, J. & File, G. "Passes, Sweeps and Visits in Attribute Grammars." *Journal of the ACM*, Vol.36, No.4, (1981), pp.841-869.

用不确定的求值函数定义了纯属性文法；用确定的求值函数定义了简单属性文法。给出了 8 类文法，并讨论了判定属于这些属性文法类别的时间复杂性。

Engelfriet, J., Rozenberg, G., & Slutzki, G. "Tree Transducers, L Systems and Two-way Machines." *Journal of Computer Systems and Software*, Vol.20 (1980), pp.150-202.

参阅关于属性文法与树处理变换程序之间关系的形式化讨论。

Even, S. *Graph Algorithms*. London: Pitman, 1979.

参阅第 1 章关于图中路径的介绍，以及第 2 章关于树的介绍，特别是 2.1 小节关于树的定义。

Ganapathi, M. & Fischer, C.N. "Description-Driven Code Generation Using Attribute Grammars." *Proceedings of the 9th Annual ACM Symposium on Principles of Programming Languages*. New York: ACM, 1982, pp.108-119.

Gokhale, M. *Parallel Evaluation of Attribute Grammars*. Technical Report No.89-17, University of Delaware, 1989.

为每一综合属性附加了权重，一旦权重超过某一阈值则衍生一个进程。

Kamimura, T. "Tree Automata and Attribute Grammars." In *Automata, Languages and Programming*, Lecture Notes in Computer Science, Vol.154. New York: Springer-Verlag, 1983, pp.374-384.

属性值作为一个固定字母表上的串处理，并将属性文法定义为从树到串的变换程序；介绍了用于树遍历的、从树到串的下推变换程序，作为属性变换程序的模型。

Kildall, G.A. *Global Expression Optimization During Compilation*. Ph.D. dissertation, University of Washington, Computer Science Group, 1972.

介绍了任意程序图上基本数据流分析的算法及其正确性证明。

Knuth, D.E. "Semantics of Context-free Languages." *Mathematical Systems Theory*, 2:2. New York: Springer-Verlag, 1968, pp.127-130.

Knuth, D.E. *The Art of Computer Programming: Vol. 1, Fundamental Algorithms*. Reading, MA: Addison-Wesley, 1973.

参阅第 143~144 页和第 213~215 页的交换缓冲算法。

Kuhn, R.H. et al. "Dependence Graphs and Compiler Optimization." *Proceedings of the 8th Annual ACM Symposium on Principles of Programming Languages*, 1981, pp.207-218.

将图抽象（结点及其边的集合被合并为单个结点）用于组织优化（该技术也用于数据流分析，参阅上述[Cocke&Schwartz, 1970]文献；以及用于在[Zelkowitz&Bail, 1974]描述的 SIMPL 优化程序中控制优化范围）。作者利用图抽象分离出一个语句集，其中的语句仅当作为一个整体处理时，才能被翻译为机器代码。

Madsen, O.L. "On the Use of Attribute Grammars in a Practical Translator Writing System." Report DAIMI, Computer Science Department, Aarhus University. 1975.

- Madsen, O.L. "Towards a Practical and General Translator Writing System." Report DAIMI, Computer Science Department, Aarhus University. 1980.
- MetaWare, Inc. *MetaWare Translator Writing System User Manual*. Santa Cruz, CA, 1981.
- Payton, T. et al. "Design Level Debugging of Attribute Grammars." Report, SDC, Burroughs, Paoli, PA, 1982.
- Pittman, T. *Practical Code Optimization by Transformational Attribute Grammars Applied to Low-Level Intermediate Code Trees*. Ph.D. dissertation, University of California at Santa Cruz, 1985.
- Pittman, T. *Using Transformational Attribute Grammars for Code Optimization*. TR-CS-86-4, Department of Computing and Information Sciences, Kansas State University, 1986.
- 讨论了 TAG, 给出中间代码树 ILT 的样例展示了 9 种声明链中的 3 种。注意采用 TAG 和 ILT 表示法会得到两大好处: (1) 分析什么是可提升的; (2) 使用 ILT 构建子表达式子树之上的运算符结点。
- Zelkowitz, M.V. & Bail, W.G. "Optimization of Structured Programs." *Software – Practice and Experience*, Vol.4, No.1, (1974), pp.51-57.
- 从里向外对结构化块进行优化。

第9章 代码生成与优化

本章旨在：

- 介绍代码生成与优化的高级课题
- 探讨循环优化与资源分配问题
- 讨论基于图着色的寄存器分配以及表达式中的寄存器分配
- 演示零地址代码如何转换为寄存器机器代码
- 在编译时的栈中模拟代码的执行
- 给出根据 IBSM 生成基于寄存器的代码的一个实例
- 演示在树文法中实现的处理器调度，它构成一个有向无环图
- 考虑面向 RISC、流水线以及向量处理器的编译程序设计

9.1 简介

迄今为止，本书中的大多数代码生成练习都是面向 Itty Bitty 栈机器，这种机器的指令集非常简单。真实的计算机并不是以这种方式建造的，理由非常简单：新奇和精彩的指令集令计算机运行更快，而高速计算机的销量比慢速计算机更好。目前还没有很好地开展对体系结构之间以及体系结构与性能之间的考虑因素的研究，大多数更好的计算机设计完全源自一些天才计算机设计人员的灵光一现，他们直觉地把握到哪些最新技术可支持人们可能正在找寻的特性。编译程序设计人员一直追求的是正交的指令集，这令他们更容易维持生计；但以速度为关键要素的代码通常采用汇编语言编写，这个市场是由速度痴迷者而不是编译程序设计人员来驱动的。

有一类重要的优化技术在很大程度上独立于计算机体系结构，但也往往无法很好地遵循基于文法的方法，这类优化技术就是循环优化。第8章描述了一些持久循环变量（PLV）、循环不变代码移动、循环的通用数据流分析等相关问题，并展示了如何以变换属性文法（TAG）表达这些分析与优化。本章更深入地钻研如何令循环执行得更快的特殊问题，包括归纳变量转换、循环展开与融合以及线性化数组（可看作是循环展开的一种特殊情况）等。公开发表的研究文献很少涉及这些概念的实现，本章也不像先前那样详尽地介绍这些实现，而是将更多的细节留待读者充分发挥自己的创造性。

本章还将描述非常规的计算机体系结构给编译程序设计人员带来的特殊问题，其中最重要的问题包括内存与寄存器分配、奇形怪状指令（又称巴洛克指令）、资源调度等通用问题。这些问题中的每一个都很难用一种基于文法的编译程序设计方法处理，但对其中的某些问题存在一些有效的折中方法，可在一个基于文法的编译程序中取得相当好的性能；同时，本章还将涉猎这些问题的其他处理方法。应注意的是，通用循环优化技术与向量计算机所要求的处理方法有相当多的重叠部分。

9.2 循环优化

大多数的计算机程序几乎将所有运行时间都花费在内循环，而这些代码仅占全部代码的10%（甚至更少）。因而，为循环优化而付出的努力会有远高出正常比例的回报。

9.2.1 循环的范围分析

在我们想到的所有针对性能回报的优化分析中，最简单的可能是将范围分析扩展到循环。其性能回报并不限于各个循环内的代码，而是延伸到整个程序；同时，还可省略全面的范围检查，以及用更短的变量和算术运算符替换程序员选用的运算符（取决于目标机器硬件）。

如前所述，除用于循环之外，标量变量的范围分析在模拟执行中是一件挺简单的事情。然而，持久循环变量的范围还依赖于循环的迭代次数，而在编译时并不是总能推断迭代的次数。本小节讨论的情况是要么迭代次数是可推断的，要么可建立迭代次数的边界。

最简单的情况显然是 **FOR** 循环结构，其中迭代次数由程序员指定的控制变量范围设置。只要起始值和终止值是常量，那么迭代次数也是常量。在少数情况下，控制变量的起始值和终止值是由同一（非常量）表达式的值进行常量置换得到；这时同样也可产生一个常量的迭代次数，只是需花费更多的编译时开销而已。例如，

```
FOR i := (n - 1) * 3 TO n * 3 + 8 DO
```

常量折叠变换将累加的常量部分向上移至表达式树的根结点，再加上消除公共子表达式的分析即有可能识别这类循环的范围。

我们已知常量表达式分析只是范围分析的一种特例，因而对于起始值和终止值之差不是一个常量值的 **FOR** 循环而言，这只是识别和建立控制变量边界的一小步。范围分析已可确定起始值和终止值的边界，起始值的下界和终止值的上界直接组成了控制变量的边界。

在其他循环结构中，分析过程更复杂一些。有时不可能推断循环迭代次数的合理边界，但此时我们关心的并不是迭代次数，而是变量的范围，后者通常有更多的限制。我们只需特别关注持久循环变量；给出这些变量的合理边界后，可采用传统方法确定其他变量的范围。

在大多数情况下（但并非总是如此），循环中持久循环变量的某个界是一个在循环之外建立的初始值。单调的持久循环变量就属这种情况，即循环中对该变量的所有赋值均令该变量按一个非负数（或非正数）递增（或递减），或乘以一个正数值（由对循环的修改表达式的范围分析确定）。除非持久循环变量在修改表达式中直接被引用，否则可能很难建立其单调性。这种分析是有价值的，因为经常会出现将一个有界的子表达式（通常是常量）直接累加到持久循环变量的情况。如果循环的控制决策是测试一个持久循环变量与一个有界表达式的关系，这就建立了其范围的另一个界，这个界通常在初始值的另一端。如果缺少一种方便的手段对循环交替执行的条件进行短路求值（例如在 *Modula-2* 语言中），程序员有时可能令持久循环变量按常量递增并设置固定的上限，以取代 **FOR** 循环结构来控制循环；对这类语言的编译程序而言，一种有益的做法可能是寻找这种情形，从而再建立迭代次数的上界。

当迭代次数有界时，一个直接或间接作为控制变量的单调持久循环变量的上界（或下界），仍可确定为迭代次数的上界与所有持久循环变量的赋值增量的上界之乘积。这同样可应用于非单调的持久循环变量，因为其下界可确定为最大负增量的上界与迭代次数的上界之乘积。

程序员引入的错误测试也可用于建立一个持久循环变量的边界，只要它们与持久循环变量的任意赋值语句的执行路径相同即可。为达到此效果，编译程序可利用的测试仅有如下形式：

```
IF v > k THEN ErrorProc
```

其中，全局数据流分析已确立 *ErrorProc* 是一个没有返回结果的过程调用，或是一个类似 **GOTO**

语句的控制结构（例如 Modula-2 语言中的 **EXIT** 语句或 C 语言中的 **break** 语句）。

迄今为止讨论的所有分析技术在编译时间上几乎都是线性的，即只需 n 次遍历整棵程序树（ n 是一个很小的固定数），再另加 m 次遍历每一嵌套层次中的每一循环体（ m 也是一个很小的固定数），就能收集到所有相关的信息。诚然，计算许多范围值时我们宁愿是保守的；但可证明这些范围值是正确且安全的，并且通常可产生有用的优化效果。对范围值的更精确限制，特别是那些不满足上述标准的变量的边界值，可能需要更复杂的理论证明以及指数级的分析时间；我们的看法是这些方法带来的性能改进还不足以弥补额外的编译时间开销。

9.2.2 归纳变量

归纳变量是一个序数变量或编译程序生成的临时变量，该变量的值依照循环的控制变量按线性关系变化。程序员定义的归纳变量在循环体中恰好被赋值一次，并且引用了该变量的所有执行路径均先经过该赋值语句，或该赋值语句是无条件的；一个临时归纳变量通常是编译程序生成的中间值，它根据控制变量（或另一归纳变量）与循环不变量的加法或乘法计算得到。尽管许多归纳变量是以此方式从控制变量派生的，但其中一个很重要的观念是它们未必这样派生。考虑如下的循环例子：

```
n := 100;
FOR i := 1 TO 10 DO
    j := i * 8 + k;
    a[j, 3] := n;
    n := n - 10
END
```

控制变量 i 显然是一个归纳变量，因而 j 也是一个归纳变量，它由 i 与循环不变量的乘法和加法派生。数组 a 的机器级下标就没有那么明显了，它是由归纳变量 j 乘以数组片的大小、再加上第二个下标得到的。在上述例子中， n 也是一个归纳变量，因为尽管 n 是独立派生的，但它的值总是 $i * (-10) + 90$ 。归纳变量分析的结果是，用于计算 j 和数组下标的迭代值的乘法，可优化为循环不变量的加法或减法，因而它们更加类似于 n 的循环计算；这里当然假设了一次加法比一次乘法（可能还跟着一次加法）的开销更少，通常情况也确实如此。

在这种情况下，编译程序将在循环之前插入 j 的初始化步骤 $j := k$ ，然后用一个（有希望更快的）加法 $j := j + 8$ 取代循环体中的赋值语句。编译程序产生的数组下标临时变量也可执行类似的变换。此时，变量具有与变换前相同的值序列，但已从循环体中消除了两条乘法语句。此外，现在很容易看出变量 j 是死的，因而循环体中对该变量的赋值语句可一并消除，如下所示：

```
n := 100;
tv := k * asize8 + abase + 3;
FOR i := 1 TO 10 DO
    tv := tv + asize8;
    a[tv] := n;
    n := n - 10
END
```

数组访问剩下的加法可合并为一种索引寻址方式，从而令性能进一步提升，而这在涉及乘法时是不可能实现的。

为识别一个归纳变量，我们须对用于常量折叠和范围分析的模拟执行数据流分析稍作改进。很容易找出一个由其他归纳变量派生的归纳变量，只要能证明该变量位于主循环的执行路径上，且不是一个持久循环变量，且其值是由一个已知的归纳变量乘以或加上循环不变量合成的。也容易判断一个仅赋值一次的持久循环变量是否归纳变量，只要看该变量的修改是否限制在与一个循环不变量的加法或减法即可。我们的目标是将所有派生的归纳变量转换为持久循环变量。在找出循环不变量和持久循环变量后，这一分析技术最多再需要一次遍历循环体，具体细节留作练习。

9.2.3 循环展开

将循环不变量移出循环后，改进循环执行时间的最简单且最有效的优化是循环展开。一个固定迭代次数的短循环可完全展开，彻底消除循环的开销，并有可能同时将常量折叠和消除公共子表达式优化应用到数组下标的计算以及涉及控制变量的其他表达式。即便我们无法完全展开一个循环，也可将循环展开一次，从而每次执行循环时处理两个迭代。这一小小的优化可将循环的有效开销减少一半，并且为将消除公共子表达式优化应用到两个迭代上提供了更多机会。

具有固定边界的 **FOR** 循环相当常见，有幸遇到这种情况时，可执行最简单、同时也是最有效的循环展开。编译程序很容易按所需的次数将循环体复制多份，并将控制变量的引用替换为常量。当下标范围超出实现人员定义的某一上限时，仍可能判断由程序确定的循环迭代次数是否偶数（或另有某一个小的约数）；只要再稍加分析，如果仍可证明其范围是一个常量或一个小常量的乘积，编译程序还可对非循环不变的下标边界进行求值。然而在大多数情况下，下标范围的复杂分析未必就是有理的，除非是为向量处理器寻找内循环向量化的机会。

对于 **FOR** 循环以外的其他循环，分析显得更加困难。归纳变量分析可能仍可推断迭代的次数，但是在程序设计语言支持这种循环结构时，期望程序员使用 **FOR** 循环或其等价形式以实现最佳性能并非没有道理。然而，部分地展开 **WHILE** 和 **REPEAT** 循环可能仍是有意义的，这取决于目标硬件。在这些情况下，迭代片断之间必须有循环终止的测试，不过尽管如此，展开后的循环为编译程序在两个片断之间消除公共子表达式提供了一些机会。代码清单 9.1 展示了如何使用 Modula-2 语言的 **LOOP-EXIT** 结构展开一个循环。

代码清单 9.1 一次展开一个 **WHILE** 循环

<pre> WHILE a < b DO b := b - a * k; a := a + 1 END;</pre>	<pre> LOOP IF a >= b THEN EXIT END; b := b - a * k; a := a + 1; IF a >= b THEN EXIT END; b := b - a * k; (* a₁ * k = a₀ * k + k *) a := a + 1 END;</pre>
--	--

将一组恰好横跨一个多维数组的嵌套循环的外循环展开，相当于将数组的引用理解成该数组好像被声明为一个一维数组一样。因而，每一次迭代时不是进行复杂的多维下标计算，而是

由单个下标横穿整个数组，从而避免了额外的乘法和加法计算步骤。这种技术称为数组线性化，因为它使得代码生成等价于一个线性（一维）数组的代码生成。然后线性化的数组循环可部分展开，以实现更多的性能改进。代码清单 9.2 仍以源代码形式展示了将一个数组线性化的效果，请注意数组 **A1** 与数组 **A** 位于相同的内存地址空间。

代码清单 9.2 线性化一个数组

<pre>VAR A: ARRAY [1 .. 9, 3 .. 17] OF REAL; FOR i := 1 TO 9 DO FOR j := 3 TO 17 DO A[i, j] := 0.0 END END END</pre>	<pre>VAR A1[ADR(A)]: ARRAY [1 .. 135] OF REAL; FOR i := 1 TO 135 DO A1[i] := 0.0 END END</pre>
---	---

9.3 寄存器与内存分配

大多数计算机为程序员提供了直接可用的几种不同类型的内存，这些内存的性能往往有相当大的差别。通常有少量非常高速且易于访问的内存，称为寄存器；另有大量较慢的内存用于保存整个程序及其数据。我们此时并不关心高速缓存这类内存，因为本质上它对程序的操作是透明的。高速缓存类似于主存，但访问速度更快，它只是将最近未用到的数据和指令转储到较慢的主存中。我们此时也不关心虚拟内存；虚拟内存使用磁盘文件模仿和扩展主存，但速率较低。虚拟内存对于运行程序而言也是透明的，因为对可用物理内存之外的引用将引发一个由操作系统软件处理的中断。

许多计算机体系结构将寄存器和主存空间进一步细分，使得其中一些子集比另一些子集更容易访问，或赋予其中某些空间特殊的意义。寄存器可划分为地址寄存器（用于保存主存的地址）和数据寄存器（用于保存中间计算结果值）。主存可划分为“段”或“页”，使得整个内存的一小部分比在整个内存中的任意引用更加易于访问。

上述细分的主要理由可用地址空间的概念来解释。使用 n 个位可区分 2^n 个不同的对象或操作，有 65 536 个位置的内存地址空间需要 16 位寻址，而 4096 个位置则需要 12 位，256 个位置仅需要 8 位。为将最多的功能压缩在一个有限的指令字长度中，我们希望限制分配给任一功能的每一指令字中的位数。如果像早期计算机那样，在两个操作数相加时，每一指令由内存中任意两个位置显式地寻址，1024 字的主存的完整地址空间必须在指令中占用 20 位。如果令其中的一个操作数为单个寄存器，则可将指令字中的地址分量减少一半。如果有多个寄存器，则寄存器也有一个地址空间：利用 3 位即可为 8 个寄存器寻址。将 1MB 内存划分为 65 536 字节的段，每次仅访问其中的一个或两个段，就可从内存访问指令中删除 3~4 位。若有 16 个通用寄存器，则每一寄存器均需 4 位寻址；在可能涉及 2~3 个寄存器的指令中，可以快速完成累加。将寄存器空间划分为独立的地址寄存器和数据寄存器，可从每一条多寄存器的指令中消除 2~3 位。

为有效利用寄存器和内存的地址空间，编译程序须决定哪些值应放入寄存器中，以及何时放入为宜。程序员看一眼即能够察觉到一段段可得益于寄存器分配的变量用法，但是由一个算法机械地得出同一结论就远不止这样简单了。C 程序设计语言由程序员自己做出抉择，实际上这可能使得编译程序尝试正确地分配寄存器的任务更加复杂。在计算机体系结构中，这一问题

可能还掺杂了各种各样的寻址模式。编译程序往往因生成低效的代码而声名狼藉，因为编译程序设计人员很容易就简单地选择一种可正确工作的寻址模式，错失了生成更小、更快代码的机会，而程序员毫不费力即可洞察到这些机会。

9.3.1 寄存器分配算法

关于最优的寄存器分配方案研究已有大量文献 [Chaitin, 1982; Chaitin 等, 1981; Chow&Hennessy, 1984; Kennedy, 1972]。将寄存器分配给常用变量的临时存储空间，可看作等价于图着色问题。活跃变量在图中表示为区域，每一个待分配的寄存器均有一种颜色。该图将按以下方式着色：水平截面不会穿过同一颜色的两个区域，如图 9-1 所示 [Chaitin, 1982]。

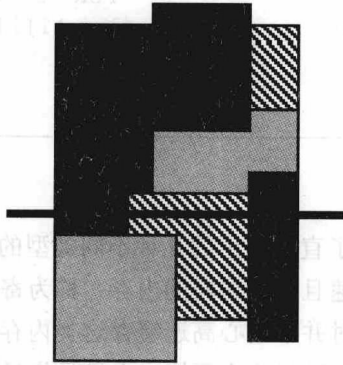


图 9-1 基于图着色的寄存器分配。水平线表示在时间轴上的某一特定时刻，有三个变量被分配到寄存器中

图着色算法为实现其目标，对寄存器分配问题的本质特征进行了假设，最值得注意的是有固定数目的寄存器可供分配，且选择哪些变量映射到寄存器是在其他地方决定的。通常，映射到寄存器的候选变量多于可用的寄存器，而过于复杂的判断会给图着色问题带来不利。此外，尽管很容易选择某一固定数目的寄存器分配给长期存储的变量，但实际上可用寄存器的真正数目还会随待求值表达式的复杂度有所变化。我们已找到一种基于已知需求的分配方案，无需复杂的图着色即可取得很好的效果，并且还更容易转换为一种基于文法的实现。

图着色并不关心表达式求值的寄存器分配问题。除了个别的例外情况，大多数现代计算机都要求算术运算或逻辑运算中至少有一个操作数必须存放在数据寄存器中。执行消除公共子表达式优化后，我们通常希望将这些部分求值结果分配到寄存器中，从而无须调用额外的内存访问操作即可在后续代码中访问它们。实现这一目标的一般做法是创建一个临时变量保存公共子表达式，然后将该变量提交给图着色或其他寄存器分配算法。

任何寄存器分配方法都必须准备好处理寄存器耗尽的问题，特别是在一个表达式的求值过程中。出现这种情况时，当前活跃的寄存器将溢出（存储）到内存中，从而腾出寄存器以供新的应用。每当执行一条有副作用的语句或调用一个有副作用的过程时，如果它们可能访问存放在寄存器中的变量，则寄存器也必须溢出。

Modula-2 和 Pascal 这一类强类型语言与 C 这一类低级语言相比，可使编译程序设计人员更容易处理这一问题。如果一种语言允许一个指针变量随便访问程序中的任何变量，正如 C 语言那样（Modula-2 语言亦同，但必须导入红色标志 `SYSTEM.ADDRESS`），编译程序实际上不可能推断当前寄存器中的变量是否安全，它们必须全部溢出到各自被指派的内存位置，并在析取指针后重新装入。而有了强类型这一特性后，编译程序不仅可安全地假定指针的析取

不会威胁任何其他类型的寄存器变量，还可假定指针析取仅会威胁动态变量，而根本不会影响任何局部变量。借助于一次遍历被调用过程的简单数据流分析，即可很容易地确定该过程是否威胁或利用了它的变参或任何非局部变量；根据该过程的寄存器需求，只有被调用过程真正用到的寄存器变量才需要溢出到内存中，并且只有那些受威胁的变量才需要在过程返回时重新装入。

高德纳对典型的（FORTRAN 语言）程序中的表达式复杂性问题进行了一些研究，发现大多数赋值语句仅涉及一个或更少的运算符，并且其中有一半只是将一个值复制给一个变量 [Knuth, 1971]。另一些研究表明，表达式求值几乎都不会复杂到需要 5 个寄存器存放中间结果的程度。将这一结论铭记在心后，许多编译程序设计人员仅允许 4 个寄存器用于表达式求值，然后在遇到一个不常见的复杂表达式时，令一个寄存器强制溢出；或者编译程序直接放弃编译，并要求程序员简化该表达式。我们的需求分配算法对表达式求值没有强加任何固定的限制（除了硬件寄存器的数目），根据上下文的需要将中间结果值或寄存器变量溢出到内存中。

已知实际应用通常将固定数目的数据寄存器分配给表达式求值，故有两种不同的方法处理剩下的寄存器。表达式求值寄存器往往被认为是易变的（volatile），即它们的值在整个过程调用中不是保持不变的。一个给定的编译程序可遵循惯例，约定其他寄存器也是易变的；或也可遵循另一更常见的惯例，约定寄存器的某一固定子集不是易变的，这一惯例要求一个过程保存并恢复该过程用到的所有非易变的寄存器。支持这一惯例的理由是基于如下假设：在活跃表达式中很少出现过程调用，而寄存器变量则可能活跃在多条语句中，包括横跨过程调用的语句。在某些情况下，操作系统调用在这方面被看作是过程调用，且不保存易变的寄存器；而在另一些情况下，系统调用还保存大多数易变的寄存器（除了那些显式地返回一个值的调用）。通常，这些惯例是由该系统的第一批编译程序编写人员确立的，后来发展的编译程序往往会遵循这些惯例。

9.3.2 表达式中的寄存器分配

一个简单的“一遍”编译程序即使不带数据流分析，也可为多寄存器机器生成合理的代码，其方法是在寄存器中模拟表达式栈。单个流动的属性可携带一个索引，表示基于寄存器的栈中值的数目；并且寄存器被排序，从而压入的第一个值总是存入（例如）寄存器 0，第二个值存入寄存器 1，如此类推。如果计算机支持寄存器到寄存器的算术运算，即可直接生成不那么高效的零地址代码，如代码清单 9.3 所示。尽管该例子展示了零地址代码如何直接转换为一种虚构的寄存器机器代码，代码生成程序也可生成单地址代码，这将消除间接寄存器的装入和存储指令，用直接的装入和存储指令取代它们以及它们相应的地址装入指令，具体细节留作练习。

代码清单 9.3 在寄存器中模拟一个零地址栈

待编译的语句: $a := (a + b) * (c - 3)$

<u>IBSM（零地址）代码:</u>	<u>寄存器机器代码:</u>	
LDC A	LDA 0, A	装入 A 的地址
LDC A	LDA 1, A	装入 A 的地址
LD	LD 1, @1	装入 A
LDC B	LDA 2, B	装入 B 的地址
LD	LD 2, @2	装入 B
ADD	ADD 1, 2	$r1 + r2 \rightarrow r1$
LDC C	LDA 2, C	装入 C 的地址
LD	LD 2, @2	装入 C
LDC 3	LD 3, #3	装入常量值 3
SUB	SUB 2, 3	$r2 - r3 \rightarrow r2$

MPY		MUL 1, 2	$r1 * r2 \rightarrow r1$
ST		ST 1, @0	存储到 A

	IBSM 代码	虚拟栈	生成的代码
		空栈	
(1)	LDC a	a 的地址	(无代码)
(2)	LDC a	a 的地址 a 的地址	(无代码)
(3)	LD	a 的值 a 的地址	(无代码)
(4)	LDC b	b 的地址 a 的值 a 的地址	(无代码)
(5)	LD	b 的值 a 的值 a 的地址	(无代码)
(6)	ADD	b 的值 寄存器 0 a 的地址	LD 0, a
(7)	ADD	寄存器 0 a 的地址	ADD 0, b
(8)	LDC c	c 的地址 寄存器 0 a 的地址	(无代码)
(9)	LD	c 的值 寄存器 0 a 的地址	(无代码)
(10)	LDC 3	常量 3 c 的值 寄存器 0 a 的地址	(无代码)
(11)	SUB	常量 3 寄存器 1 寄存器 0 a 的地址	LD 1, C
(12)	SUB	寄存器 1 寄存器 0 a 的地址	SUB 1, #3
(13)	MPY	寄存器 0 a 的地址	MUL 0, 1
(14)	ST	空栈	ST 0, a

图 9-2 编译时的虚拟栈（斜体 IBSM 运算符表示延续上一步的操作）

图 9-2 展示了一个结构更强的栈属性；在编译时，该虚拟栈中每一单元表示这一个值在运行时的物理位置。代码生成程序将变得稍微复杂一些，这是因为我们在编译时的栈中模拟代码的执行，而不是像先前那样直接生成装入一个寄存器值的代码。仅当虚拟栈的信息变得太复杂时，或仅当结果要存回目标变量时，才会生成真正的代码。跟踪图 9-2 的步骤序列，有助于理解在编译一条赋值语句 $a := (a + b) * (c - 3)$ 时是如何工作的。

第 1 步从一个空的虚拟栈开始，本应由编译程序早期版本产生的 Itty Bitty 栈机器代码，此时仅导致一个单词“a 的地址”被压入（编译时的）虚拟栈中，而不产生任何寄存器代码。第 2 步重复这一过程；第 3 步识别 IBSM 指令 LD 后，用单词“a 的值”取代虚拟栈的栈顶元素，仍不产生任何代码。接下来的两步在处理变量 b 时重复了这一过程。

第 6 步要求模拟执行的计算机将栈顶的两个元素求和，但这却无法表示在虚拟栈中；更何况该计算机也不可能不将其中一个变量放入寄存器就实现两个内存中的值求和。因而，此时将产生寄存器代码，从内存中装入到寄存器 0；第 7 步则再次考虑 IBSM 指令 ADD，产生从内存到寄存器的 ADD 指令。更新后的虚拟栈表明栈顶元素是存放在寄存器 0 中的值。

第 8~12 步重复相同的过程处理第二个子表达式 $(b - 3)$ 。注意，常量值显式地表示在虚拟栈中。如果编译程序先前未执行常量折叠变换，此处也可能实现一些常量折叠，因为代码生成程序可检测到为虚拟栈中两个常量的算术运算生成代码的意图，此时直接以一个适当的常量单词取代栈顶元素即可，而不必生成任何代码。

最后两步继续模拟过程，分别生成适当的寄存器代码。容易看出生成的目标机器代码往往都非常好，并且在许多情况下还是最优的（正如本例所示）。

基于模拟执行生成寄存器代码的关键数据结构是虚拟栈，其实现既可以是一个由单元组成的链表，也可以是一个由单元组成的数组。每一单元是一个记录，其中有一个标记字段表示所包含数据的类别（变量地址、值、常量或寄存器），还有一个变体表示变量引用的数目或地址、常量值、或寄存器数目。由于并非所有栈单元的值都会表示在一个寄存器中（注意图 9-2 中的栈最多伸展到 4 个单元，但只需要 2 个寄存器），所以有必要携带一个可用寄存器的集合，在有需要时从中挑选一个使用。

如果需要一个新的寄存器时，可用寄存器集合为空，则某一个在用的寄存器必须溢出到内存中。若不借助向后数据流分析为选择寄存器提供信息，一个最明显的选择就是虚拟栈中最深层的那个寄存器，因为该寄存器中的值离再次被请求使用的时间最长。当该寄存器溢出到内存时，它在虚拟栈中的单元被一个指向存放它的临时内存变量的值引用所替代，同时生成一条存储指令。如果该栈单元再次到达栈顶，算法将自动地重新装入（或在一条算术运算指令中直接引用内存）。

代码清单 9.4 展示了一个模块的实现，它根据作为参数传递的 IBSM 运算符生成寄存器代码。这些代码并不是在所有基于寄存器的体系结构中都可完美地工作，但采用表 9-1 列出的常量，它确实可以为四种流行的计算机生成令人满意的（尽管并不总是最优的）代码。数据表的数值码可参阅附录 D。

代码清单 9.4 一个根据 IBSM 生成基于寄存器的代码的模块

```
MODULE CodeConstants;  
(* 用于一个将 IBSM 转换为处理通用寄存器 CPU 的翻译程序 *)
```

```

EXPORT
    GenericOpcode,                (* 操作码的枚举列表 *)
    AddressMode;                  (* 寻址模式的枚举列表 *)

TYPE
    AddressMode = (
        non,                       (* 无寻址或与寻址无关 *)
        con,                       (* 直接常量 *)
        mem,                       (* 内存引用 *)
        ind,                       (* 寄存器间接寻址 *)
        reg,                       (* 寄存器中的值 *)
        ccc                        (* 比较的结果 *)
    );
    GenericOpcode = (
        Nop, Load, Store,         (* 内存与寄存器之间的交换 *)
        Cmpr, Add, Subt, Mlpy, Neg, (* 算术运算和比较运算 *)
        Andd, Orr, Jump, Bcc, BackP (* 条件、分支和逻辑运算 *)
    );
END CodeConstants;

MODULE TargetMachine;
(* 定义了一个特定 CPU 的数据表 *)

FROM CodeConstants IMPORT
    GenericOpcode,                (* 操作码的枚举列表 *)
    AddressMode;                  (* 寻址模式的枚举列表 *)

EXPORT
    CPU,                          (* 一个表示目标处理器的通用数字 *)
    BigEndian,                    (* 内存字节的含义: true 表示最高有效位在最前 *)
    RelBranch, HalfAdd,           (* 分支选项: true 表示关系或移位运算 *)
    HasImmMode,                   (* true 表示内嵌常量的寻址模式 *)
    WeirdMulti,                   (* 在多种不同的乘法实现中选择一种 *)
    Register,                     (* 一个包含所有寄存器的整数子界 *)
    LoReg, HiReg,                 (* 可用寄存器编号的范围 *)
    AdReg,                        (* 用于内存寻址的寄存器; 若都可用则等于 LoReg *)
    ConditionCodes,               (* 测试比较结果的 6 个编码 *)
    OpCodeIndex, OpCodeTable;     (* 某一特定 CPU 的数据表 *)

CONST
    (* 表中的值请参阅表 9-1 和附录 D *)

TYPE
    Register = [0 .. TopReg];
    TableRange = [0 .. EndTable];

VAR
    (* 这些实际上是常量表, 而不是变量 *)
    ConditionCodes: ARRAY [0 .. 5] OF CARDINAL;
    (* 每一入口测试一种比较的结果: <=, <, =, <>, >=, > *)
    OpCodeIndex: ARRAY GenericOpcode, AddressMode OF TableRange;
    (* 每一入口指向 OpCodeTable 中一个序列的起始位置 *)

```

```

OpCodeTable: ARRAY TableRange OF [0 .. 255];
  (* 每一操作码由如下 6 个以上的字节的序列组成:
      TotalBytes      - 该指令的字节数
      RegByte         - 一个寄存器编号的字节位置, 或比较结果
      RegPosn         - 用于定位它的乘数
      AddByte         - 地址的首字节, 或第 2 个寄存器, 或比较结果
      NumAddBytes     - 地址的字节数, 或寄存器、比较结果的位置
      NumOpBytes      - 跟在后面的操作码的字节数
      opbytes ...     - 操作码的字节
  *)

BEGIN
  (* 以某种方式装入数据表 *)
END TargetMachine;

(* 输出过程 *)

MODULE TargetGenerator;
(* 基于表格解释的内存中代码生成程序 *)

FROM TargetMachine IMPORT
  BigEndian,                (* 内存字节的含义: true 表示最高有效位在最前 *)
  Register,                 (* 一个包含所有寄存器的整数子界 *)
  OpCodeIndex, OpCodeTable; (* 某一特定 CPU 的数据表 *)

FROM CodeConstants IMPORT
  GenericOpCode,            (* 操作码的枚举列表 *)
  AddressMode;              (* 寻址模式的枚举列表 *)

FROM Somewhere IMPORT
  maxCode;                  (* 支持生成的目标代码的最大空间 *)

EXPORT
  EmitTarget;               (* 输出一条目标机器指令 *)

VAR
  ObjectCode: ARRAY [0 .. maxCode] OF [0 .. 255];

PROCEDURE EmitTarget(theOp: GenericOpCode; aMode: AddressMode; regNum,
  opAddress: INTEGER; Location: CARDINAL; VAR NextLoc: CARDINAL);
VAR
  index, opdata, temp, opAdd: CARDINAL;
  regNo: Register;
BEGIN
  IF theOp = BackP THEN      (* 回填一个分支 *)
    NextLoc := Location;
    opdata := OpCodeIndex[Bcc, mem] + 6;
    Location := NextLoc - OpCodeTable[opdata]
  ELSE                      (* 普通指令的代码 *)
    opdata := OpCodeIndex[theOp, aMode];
    NextLoc := Location + OpCodeTable[opdata];
    temp := OpCodeTable[opdata + 5];
  
```

```

FOR index := Location + temp TO NextLoc - 1 DO
    ObjectCode[index] := 0 (* 清除未指定的字节 *)
END;
opdata := opdata + 6;
FOR index := 0 TO temp - 1 DO (* 插入操作码的字节 *)
    ObjectCode[Location + index] := OpCodeTable[opdata + index]
END;
IF aMode > non THEN (* 插入寄存器引用 *)
    regNo := regNum;
    index := OpCodeTable[opdata - 5];
    temp := regNo * OpCodeTable[opdata - 4];
    ObjectCode[Location + index] := ObjectCode[Location + index] + temp MOD 256;
    IF temp > 255 THEN
        IF BigEndian THEN
            ObjectCode[Location + index - 1] := ObjectCode[Location + index - 1]
                + temp DIV 256
        ELSE
            ObjectCode[Location + index + 1] := ObjectCode[Location + index + 1]
                + temp DIV 256
        END
    END
END
END (* IF aMode > non *)
END; (* IF theOp = BackP *)
IF (aMode = mem) OR (aMode = con) THEN (* 插入地址 *)
    temp := Location + OpCodeTable[opdata - 3] - 1;
    IF BigEndian THEN
        FOR index := OpCodeTable[opdata - 2] TO 1 BY -1 DO
            opAdd := opAddress MOD 256;
            ObjectCode[temp + index] := ObjectCode[temp + index] + opAdd;
            opAddress := opAddress DIV 256
        END
    ELSE
        FOR index := 1 TO OpCodeTable[opdata - 2] DO
            opAdd := opAddress MOD 256;
            ObjectCode[temp + index] := ObjectCode[temp + index] + opAdd;
            opAddress := opAddress DIV 256
        END
    END
END (* IF BigEndian *)
ELSIF (aMode = reg) OR (aMode = ind) THEN (* 插入第二个寄存器 *)
    regNo := opAddress;
    index := OpCodeTable[opdata - 3];
    ObjectCode[Location + index] := ObjectCode[Location + index] + regNo * OpCodeTable
        [opdata - 2]
END (* IF (aMode = mem) OR (aMode = con) *)
END EmitTarget;

END TargetGenerator;

(* 输出过程 *)

MODULE CodeGenerator;
(* 一个将 IBSM 转换为通用寄存器 CPU 的翻译程序 *)

```

```

FROM TargetMachine IMPORT
    RelBranch, HalfAdd,
    HasImmMode,
    WeirdMulti,
    Register,
    LoReg, HiReg,
    AdReg,
    ConditionCodes;
(* 分支选项: true 表示关系或移位运算 *)
(* true 表示内嵌常量的寻址模式 *)
(* 在多种不同的乘法实现中选择一种 *)
(* 一个包含所有寄存器的整数子界 *)
(* 可用寄存器编号的范围 *)
(* 用于内存寻址的寄存器;若都可用则等于 LoReg *)
(* 测试比较结果的 6 个编码 *)

FROM TargetGenerator IMPORT
    EmitTarget;
(* 输出一条目标机器指令 *)

FROM CodeConstants IMPORT
    GenericOpcode,
    AddressMode;
(* 操作码的枚举列表 *)
(* 寻址模式的枚举列表 *)

FROM Somewhere IMPORT
    AllocTempVar, ReleaseTempVars,
    Deepest;
(* 临时变量的分配 *)
(* 虚拟栈的最大深度 *)

EXPORT
    EmitIBSM, BackPatch;

(* 局部定义 *)

TYPE
    regset = SET OF Register;
    stackrange = [0 .. Deepest];
    stackCell = RECORD
        itsType: AddressMode;
        isNegative: BOOLEAN;
        itsValue: INTEGER
    END;

VAR
    queuedOpcode: CARDINAL;
    busyRegisters: regset;
    virtualStack: ARRAY stackrange OF stackCell;
    topStack: stackrange;
    CurrentLoc, hotcc: CARDINAL;
(* 先前的调用留下待回填的 LDC 操作码 *)
(* 当前正在使用的寄存器 *)

(* 作为工具使用的过程 *)

PROCEDURE PushVirtual(kind: AddressMode; datum: INTEGER);
BEGIN
    INC(topStack);
    WITH virtualStack[topStack] DO
        itsType := kind;
        itsValue := datum;
        isNegative := FALSE
    END
END PushVirtual;

```

```

PROCEDURE PopVirtual(): INTEGER;
BEGIN
    DEC(topStack);
    RETURN virtualStack[topStack + 1].itsValue
END PopVirtual;

PROCEDURE StoreRegister(theReg: Register; theAddress: INTEGER);
VAR
    temp: INTEGER;
BEGIN
    EmitTarget(Store, mem, theReg, theAddress, CurrentLoc, CurrentLoc)
END StoreRegister;

PROCEDURE SpillRegister(theReg: INTEGER);
VAR
    temp: INTEGER;
    here: stackrange;
    gotit: BOOLEAN;
BEGIN
    gotit := FALSE;
    FOR here := 1 TO topStack DO
        WITH virtualStack[here] DO
            IF (itsType = reg) AND (itsValue = theReg) THEN
                IF NOT gotit THEN
                    temp := AllocTempVar();           (* 将该寄存器溢出到内存 *)
                    EmitTarget(Store, mem, theReg, temp, CurrentLoc, CurrentLoc);
                    gotit := TRUE
                END;
                itsType := mem;
                itsValue := temp
            END (* IF *)
        END (* WITH *)
    END (* FOR *)
END SpillRegister;

PROCEDURE CantUseIt(Taken, Weird: BOOLEAN; theReg: Register): BOOLEAN;
BEGIN
    RETURN Taken
        OR Weird AND (WeirdMulti = 2) AND NOT ODD(theReg)
        OR Weird AND (WeirdMulti = 3) AND (theReg > LoReg)
END CantUseIt;

PROCEDURE GetRegister(Weird: BOOLEAN): Register;
VAR
    areg: Register;
    hear, there: stackrange;
    usage: ARRAY [LoReg .. HiReg] OF CARDINAL;
BEGIN
    areg := LoReg;                                     (* 获取一个寄存器 *)
    WHILE (areg < HiReg) AND CantUseIt(areg IN busyRegisters, Weird, areg) DO
        INC(areg)
    END;
    IF NOT (areg IN busyRegisters) THEN
        INCL(busyRegisters, areg);
        RETURN areg
    END

```

```

END;
FOR hear := 1 TO topStack DO
  WITH virtualStack[hear] DO
    IF itsType = reg THEN          (* 在栈中寻找任一寄存器 *)
      areg := itsValue;
      IF NOT CantUseIt(FALSE, Weird, areg) THEN
        SpillRegister(areg);      (* 将该寄存器溢出到内存 *)
        RETURN areg
      END
    END (* IF *)
  END (* WITH *)
END (* FOR *)
END GetRegister;

PROCEDURE LoadThis(which: stackrange; prefereg: BOOLEAN);
VAR
  areg: Register;
BEGIN
  WITH virtualStack[which] DO
    IF (itsType <> reg) OR CantUseIt(FALSE, prefereg, itsValue) THEN
      areg := GetRegister(prefereg);
      EmitTarget(Load, itsType, areg, itsValue, CurrentLoc, CurrentLoc);
      itsType := reg;
      itsValue := areg
    END
  END
END LoadThis;

PROCEDURE Flushcc;                (* 将不是立即使用的比较结果保存起来 *)
BEGIN
  IF hotcc > 0 THEN
    LoadThis(hotcc, FALSE)
  END;
  hotcc := 0
END Flushcc;

PROCEDURE SwapMaybe(always: BOOLEAN): BOOLEAN;    (* 如果交换则返回 true *)
VAR
  tempCell: stackCell;
BEGIN
  IF NOT always THEN                (* 若栈顶是寄存器且其下元素不是，则不交换 *)
    IF (virtualStack[topStack - 1].itsType <> reg)
      AND (virtualStack[topStack].itsType = reg) THEN
      RETURN FALSE
    ELSIF (virtualStack[topStack - 1].itsType <> reg)
      OR (virtualStack[topStack].itsType = reg) THEN
      IF virtualStack[topStack - 1].isNegative
        OR NOT virtualStack[topStack].isNegative THEN
        RETURN FALSE
      END
    END
  END
  tempCell := virtualStack[topStack];
  virtualStack[topStack] := virtualStack[topStack - 1];
  virtualStack[topStack - 1] := tempCell;

```

```

    RETURN TRUE
END SwapMaybe;

PROCEDURE DoOpcode(theOp: GenericOpcode);
VAR
    areg: Register;
    wasNeg: BOOLEAN;
BEGIN
    Flushcc;
    IF SwapMaybe(FALSE) THEN END;
    LoadThis(topStack, (theOp = Mlpy) AND (WeirdMulti > 1));
    wasNeg := virtualStack[topStack].isNegative;
    areg := PopVirtual();
    WITH virtualStack[topStack] DO
        IF theOp = Add THEN
            IF wasNeg <> isNegative THEN
                theOp := Subt
            END
        ELSIF theOp = Mlpy THEN
            wasNeg := wasNeg <> isNegative;
            IF (WeirdMulti = 3) AND ((itsType <> reg) OR (itsValue <> HiReg)) THEN
                SpillRegister(HiReg)          (* 86 型 CPU 的寄存器是一种特殊情况 *)
            END
        END;
        EmitTarget(theOp, itsType, areg, itsValue, CurrentLoc, CurrentLoc);
        isNegative := wasNeg;
        itsType := reg;
        itsValue := areg
    END
END DoOpcode;

PROCEDURE Compare(ccResult: CARDINAL);
VAR
    areg: Register;
    wasNeg: BOOLEAN;
BEGIN
    Flushcc;
    IF NOT SwapMaybe(virtualStack[topStack - 1].isNegative
        AND NOT virtualStack[topStack].isNegative) THEN
        ccResult := 5 - ccResult
    END;
    LoadThis(topStack, FALSE);
    wasNeg := virtualStack[topStack].isNegative;
    areg := PopVirtual();
    WITH virtualStack[topStack] DO
        IF wasNeg <> isNegative THEN          (* 符号必须相同 *)
            EmitTarget(Neg, reg, areg, 0, CurrentLoc, CurrentLoc)
        END;
        EmitTarget(Cmpr, itsType, areg, itsValue, CurrentLoc, CurrentLoc);
        isNegative := FALSE;
        itsType := ccc;
        IF wasNeg AND (ccResult DIV 2 <> 1) THEN
            itsValue := (ccResult + 4) MOD 8      (* 为负数则反向比较 *)
        ELSE
            itsValue := ccResult
        END
    END
END Compare;

```



```

    END
  END
END Compare;

PROCEDURE DoLoad();
VAR
  offset, cond: INTEGER;
  areg: Register;
  theOp: GenericOpcode;
BEGIN
  WITH virtualStack[topStack] DO
    IF itsType = con THEN
      itsType := mem
    ELSE
      LoadThis(topStack, FALSE);
      areg := PopVirtual();
      IF virtualStack[topStack + 1].isNegative THEN
        EmitTarget(Neg, reg, areg, 0, CurrentLoc, CurrentLoc)
      END;
      EmitTarget(Load, ind, areg, areg, CurrentLoc, CurrentLoc)
    END
  END
END DoLoad;

PROCEDURE DoBranch();                                (* 总是产生长地址 *)
VAR
  offset, aLoc, cond: INTEGER;
  theOp: GenericOpcode;
BEGIN
  offset := PopVirtual();
  IF virtualStack[topStack].itsType = con THEN
    IF PopVirtual() = 0 THEN
      theOp := Jump                                (* 无条件跳转 *)
    ELSE
      RETURN                                        (* 无分支 *)
    END
  ELSE
    theOp := Bcc;
    IF virtualStack[topStack].itsType <> ccc THEN
      LoadThis(topStack, FALSE);
      PushVirtual(con, 0);
      Compare(3)                                    (* 比较栈顶是否不等于 0 *)
    END;
    cond := PopVirtual()
  END;
  IF NOT RelBranch THEN
    aLoc := CurrentLoc;
    offset := offset + aLoc
  END;
  IF HalfAdd THEN
    offset := offset DIV 2
  END;
  EmitTarget(theOp, mem, cond, offset, CurrentLoc, CurrentLoc)
END DoBranch;

```

```

PROCEDURE EmitIBSM(datum: INTEGER; inAddress: CARDINAL; VAR outAddress: CARDINAL);
VAR
    areg: Register;
BEGIN
    CurrentLoc := inAddress;
    IF queuedOpcode = 0 THEN
        queuedOpcode := datum
    ELSE
        PushVirtual(con, datum);
        queuedOpcode := queuedOpcode DIV 32
    END;
    WHILE (queuedOpcode > 0) AND (queuedOpcode MOD 32 <> 28) DO
        CASE queuedOpcode MOD 32 OF
            1:
                DoBranch() |
            8:
                (* DUPE 指令 *)
                WITH virtualStack[topStack] DO
                    PushVirtual(itsType, itsValue)
                END |
            9:
                (* SWAP 指令 *)
                IF SwapMaybe(TRUE) THEN
                    END |
            11:
                (* MPY 指令 *)
                DoOpcode(Mlpy) |
            12:
                (* ADD 指令 *)
                DoOpcode(Add) |
            14:
                (* OR 指令 *)
                DoOpcode(Orr) |
            15:
                (* AND 指令 *)
                DoOpcode(Andd) |
            16:
                (* EQUAL 指令 *)
                Compare(2) |
            17:
                (* LESS 指令 *)
                Compare(1) |
            18:
                (* GRTR 指令 *)
                Compare(5) |
            20:
                (* NEG 指令 *)
                WITH virtualStack[topStack] DO
                    isNegative := NOT isNegative
                END |
            26:
                (* ST 指令 *)
                Flushcc;
                LoadThis(topStack, FALSE);
                areg := PopVirtual();
                WITH virtualStack[topStack] DO
                    EmitTarget(Store, mem, areg, itsValue, CurrentLoc, CurrentLoc)
                END |
            27:
                (* LD 指令 *)
                DoLoad() |
            29:
                (* NIBL 指令 *)
                queuedOpcode := queuedOpcode DIV 32;
                PushVirtual(con, queuedOpcode MOD 32) |
            30, 31:

```

```

        PushVirtual(con, queuedOpcode MOD 2)
    END; (* CASE *)
    queuedOpcode := queuedOpcode DIV 32
END; (* WHILE *)
outAddress := CurrentLoc
END EmitIBSM;

PROCEDURE BackPatch(offset: INTEGER; inAddress: CARDINAL);
VAR
    aLoc, cond: INTEGER;
BEGIN
    IF NOT RelBranch THEN
        aLoc := inAddress + offset
    ELSE
        aLoc := offset
    END;
    IF HalfAdd THEN
        aLoc := aLoc DIV 2
    END;
    EmitTarget(BackP, mem, 0, aLoc, inAddress, inAddress)
END BackPatch;

(* CodeGenerator 初始化代码 *)
BEGIN
    topStack := 0;
    busyRegisters := regset{};
    queuedOpcode := 0;
    hotcc := 0
END CodeGenerator;

```

表 9-1 一些常见计算机型号的目标机器常量

CONST CPU	=	11;	86;	370;	68000;
BigEndian	=	FALSE;	FALSE;	TRUE;	TRUE;
HasImmMode	=	TRUE;	TRUE;	FALSE;	TRUE;
RelBranch	=	TRUE;	TRUE;	FALSE;	TRUE;
HalfAdd	=	TRUE;	FALSE;	FALSE;	FALSE;
WeirdMulti	=	2;	3;	0;	1;
LoReg	=	0;	0;	0;	0;
HiReg	=	4;	3;	12;	7;
TopReg	=	7;	7;	15;	15;
AdReg	=	0;	3;	0;	0;
FramePtrReg	=	5;	6;	13;	14;
EndTable	=	208;	203;	156;	236;
ConditionCodes:		7, 5, 3, 2, 4, 6;	14, 12, 4, 5, 13, 15;	13, 4, 8, 7, 11, 2;	15, 13, 7, 6, 12, 14;

根据 IBSM 的定义，而不是图 9-2 所示的逻辑，Modula-2 语言的代码在一个变量被取出或存入之前，并不区别局部变量地址和常量。这些代码令人想起在一个基于 TAG 的代码生成程序中的语义动作例程，或者是作为代码生成时调用的子例程合并到一个手工编写的编译程序中。代码清单 9.5 的 TAG 片段展示了这些语义动作例程可能是如何被使用的。注意，该语法尽可能地遵循了 Itty Bitty 栈机器的形式化定义，并由代码生成程序将它转换为合适的寄存器代码。一个投入生产用的编译程序一般会定义比 IBSM 更好的中间伪码。

代码清单 9.5 在一个树平展文法中使用寄存器生成代码

 Flatten ↓inAddr:int ↑outAddr:int

- <IF expn left rite>
 expn:Flatten ↓inAddr ↑exAddr { 生成代码对布尔表达式求值 }
 [EmitIBSM ↓28 ↓exAddr ↑conAddr] { 生成 LDC 指令 }
 [EmitIBSM ↓0 ↓conAddr ↑brfAddr] { 准备待回填的地址 }
 [EmitIBSM ↓1 ↓brfAddr ↑preAddr] { 生成 BRF 指令 }
 left:Flatten ↓preAddr ↑midAddr { 生成左子树的代码 }
 [EmitIBSM ↓30 ↓midAddr ↑zerAddr] { 生成 ZERO 指令 }
 [EmitIBSM ↓28 ↓zerAddr ↑konAddr] { 生成 LDC 指令 }
 [EmitIBSM ↓0 ↓konAddr ↑brAddr] { 准备待回填的地址 }
 [EmitIBSM ↓1 ↓brAddr ↑finAddr] { 生成 BRF 指令 }
 [BackPatch ↓finAddr-preAddr ↓brfAddr] { 回填第一条 BRF 指令 }
 rite:Flatten ↓finAddr ↑outAddr { 生成右子树的代码 }
 [BackPatch ↓outAddr-finAddr ↓brAddr] { 回填第二条 BRF 指令 }
- <Assn theVar expn>
 expn:Flatten ↓inAddr ↑midAddr { 生成代码对表达式求值 }
 theVar:Flatten ↓midAddr ↑postAddr { 访问栈中变量的代码 }
 [EmitIBSM ↓9 ↓postAddr ↑finAddr] { SWAP 指令 (无代码) }
 [EmitIBSM ↓26 ↓finAddr ↑outAddr] { 生成 ST 指令 }
- <Less left rite>
 left:Flatten ↓inAddr ↑midAddr { 生成左子表达式的代码 }
 rite:Flatten ↓midAddr ↑postAddr { 生成右子表达式的代码 }
 [EmitIBSM ↓17 ↓postAddr ↑outAddr] { 生成 LESS 指令 }
- <Plus left rite>
 left:Flatten ↓inAddr ↑midAddr { 生成左子表达式的代码 }
 rite:Flatten ↓midAddr ↑postAddr { 生成右子表达式的代码 }
 [EmitIBSM ↓12 ↓postAddr ↑outAddr] { 生成 ADD 指令 }
- <Star left rite>
 left:Flatten ↓inAddr ↑midAddr { 生成左子表达式的代码 }
 rite:Flatten ↓midAddr ↑postAddr { 生成右子表达式的代码 }
 [EmitIBSM ↓11 ↓postAddr ↑outAddr] { 生成 MPY 指令 }
- <VAR>%offset
 [EmitIBSM ↓28 ↓inAddr ↑midAddr] { 生成 LDC 指令, }
 [EmitIBSM ↓offset ↓midAddr ↑outAddr] { 及其地址常量 }
- <Fetch expn>
 expn:Flatten ↓inAddr ↑midAddr { 生成地址表达式的代码 }
 [EmitIBSM ↓27 ↓midAddr ↑outAddr] { 生成 LD 指令 }
- <CON>%value
 [EmitIBSM ↓28 ↓inAddr ↑midAddr] { 生成 LDC 指令, }
 [EmitIBSM ↓value ↓midAddr ↑outAddr] { 及其常量 }
-

9.3.3 更好的寄存器分配数据流分析

如前所述,模拟执行是一种向前的数据流分析形式,可为寄存器分配提供信息,使得寄存器分配可以远远优于朴素的基于寄存器的栈方案。向后的数据流分析可找出应保存在寄存器中的候选中间值,从而进一步减少寄存器的重新装入。第8章介绍了简单的活跃变量分析,它可将活跃变量集一路附加在中间代码中,从而在生成目标机器代码时有可能确定任一计算结果值是否可能被复用。

如果每一变量的信息根据距离度量其“年龄”并加设使用频率,那么这些信息会变得更加有用。这时不再是为每一变量传播单个位(表示变量是活跃的或死的),而是携带一个小数字(可能是[0..256]范围内的一个小整数)作为“寄存器首选值”。在向后的数据流分析中,每次引用一个变量都导致首选值递增一个常量,譬如8或16(但不会超过最大值);每次变量不再使用时,首选值递减1(但不会小于1)。在每条语句或变量引用中的所有非0值均可任意递减,因为被引用的变量导致的递增将多于递减。可调整增量值和递减频率的选择,以取得最佳效果。

这些信息将附加到一个基本块的中间代码树上,使得在一次向前遍历并生成代码时,每当有一个计算得到的结果值存储到一个指定变量或从该变量装入一个值时,如果该变量是活跃的,则将结果也保存在寄存器中。如果要保存的值多于可用的寄存器,变量根据后续引用的邻近度和频率划分等级,以决定哪些结果将溢出到内存中。

在基本块的边界,特别是两个基本块的汇合点,这种过于简化的数据流分析遍历必须清除寄存器首选值的集合,因为另一执行路径可能为同一变量分配不同的寄存器。

将活跃变量分析用于减少寄存器的装入和存储操作时,每一寄存器都关联着一个变量的列表,该寄存器作为这些变量的一个有效副本;每一寄存器还关联着一个标志,表明该寄存器是内存的复制品还是一个仅有的副本。在代码生成时,一个存储到内存的操作的模拟执行只是将目标变量添加到该寄存器的列表中(同时将它标记为一个仅有的副本),并将该变量从所有其他寄存器的列表中删除,不产生任何机器代码。一个装入操作的模拟执行则查找寄存器集合,从中找出一个在其列表中包含该变量的寄存器;仅当找不到这样的寄存器时,才分配一个新的寄存器,并真正产生一条装入指令。一条算术运算或逻辑运算若涉及一个仅有副本的值,并且该值仍是活跃的,则必须为该运算分配另一个寄存器,否则要先存储该寄存器的内容以更新这些变量。如果一个过程调用使用了寄存器中的变量,也必须强制将这些变量的值溢出。

将存储操作推迟到真正需要该寄存器或其影子变量时才执行,往往可将存储操作全部消除,例如变量仅有的活跃引用在该寄存器不得不溢出之前已被表达式使用。尽管这一启发式方法通常并不能证明是最优的(但图着色算法可证明是最优的),但可证明它不会差于不带向前看的算法(例如代码清单9.4),并且通常与图着色算法效果一样好,但它的时间复杂度只是线性时间。注意,即使不借助于向后数据流分析确定寄存器溢出的优先级,将寄存器存储推迟到有需要时才执行仍可产生明显更好的代码。

9.3.4 循环的寄存器分配

程序中的循环,特别是小的内循环,是寄存器优化的主要候选对象,因为这里最容易感觉到消除内存装入和存储操作带来的少许速度提高。所有活跃变量都是保存到寄存器中的候选对象;当活跃变量多于寄存器时,通过遍历该循环而计算得到的寄存器首选值特别有用。

持久循环变量需要进行特殊的处理,因为一个基本原则是:循环中对每一持久循环变量的

最后赋值都应让该变量留在它在循环入口处驻留的同一寄存器中。类似地，如果一个寄存器的值在整个循环过程中都活跃（即不是一个持久循环变量），该寄存器必须溢出以供更高级别的优先级使用，并且该值必须重新装入到它在循环入口处驻留的同一寄存器中。一种更简单的（也可能是更受欢迎的）做法是，避免将在循环过程中必须溢出和重装的变量预先装入任何寄存器中，将寄存器留给那些在整个循环执行过程中一直可驻留的变量。

向后数据流分析针对持久循环变量的一种用法是选择一个目标寄存器，然后用指派的寄存器标明该持久循环变量的最终计算结果；只要有一个合适的公共累加器，就将这一选择结果向后传播给中间计算结果。然后不再采用投机的寄存器分配策略，而是由寄存器分配程序识别预先选定的寄存器，并在各个候选寄存器中选择它。向后数据流分析应在计算之前，连续选择中间寄存器的值，使得指派的寄存器在需要时不会被其他值占用。表 9-2 展示了一个例子，描述这一策略是如何工作的。相同的启发式方法还可用于一个分叉的汇合点上寄存器变量的分配，或用于在寄存器中计算传递给一个过程的参数。

表 9-2 为持久循环变量预分配寄存器。空心数字 1~4 记载了向后数据流分析的决策，实心数字 5~6 指明在基于（向前）模拟执行的代码生成过程中的后续分配决策；箭头展示了两遍过程的寄存器分配信息流动

源代码	步 骤	分配决策	数据流
agedsum := 0;	④	agedsum 是一个持久循环变量。	仍分配到 R4
FOR i := 1 TO 10 DO	③	i 是一个持久循环变量，仍分配到 R5。	↑
temp := agedsum * 0.9;	5	此行之后 agedsum 不再活跃，因而使用其寄存器计算 temp；如果 temp 后续不再用于计算 agedsum 的下一个值，或 temp 自那以后仍活跃，寄存器那时无需额外开销即可溢出。	R4 R4 ↑ ↓ ↑ ↓ ↑
agedsum := temp + val[i]	②	这一行注销了 agedsum，但保存其寄存器；不分配任何寄存器给用于计算其值的变量 temp。	↓ ↑ ↓ ↑ ↓ ↑
	6	在代码生成时，temp 仍在寄存器中，并且可立即用于 agedsum 的计算。temp 不再活跃，因而溢出是不必要的。	R4 ↑ ↓ ↑ R4 ↑ ↑
END;	①	数据流分析从此处开始。i 也是一个持久循环变量，分配到寄存器 R5。	agedsum 分配到 R4

9.3.5 寻址模式

具有多种不同寻址模式的计算机给编译程序带来的问题，通常不同于我们在寄存器分配中看到的这类问题，这是由于寄存器的数目通常远远少于一种简短的寻址模式可访问的内存位置数目。处理不同寻址模式的常见方法是为全局或局部变量随意选择自己喜欢的地址空间，然后在程序的数据空间超出寻址模式的能力时，退回到没有那么紧凑的寻址模式。只要稍作分析，编译程序即可计算每一变量的引用数目（首选内循环，或首选由程序插装返回的统计结果），然后在内存中合理分布变量，以减少慢速访问的总数。

一些计算机为算术运算提供了寄存器到内存、也可能还有内存到内存的寻址模式，以及先前我们已考虑过的内存到寄存器模式。为这些计算机生成高质量的代码需扩展虚拟栈的数据结构，以容纳那些带单个运算符与两个操作数的表达式。虚拟栈的两个顶部元素不含运算符时，

一条算术运算指令的模拟执行只是将这两个元素组合为一条简单的表达式，如图 9-3 的赋值语句 `a := b + a` 例子所示。

	IBSM 代码	虚拟栈	生成的代码									
		空栈										
(1)	LDC a	<table><tr><td>a 的地址</td><td></td><td></td></tr></table>	a 的地址			(无代码)						
a 的地址												
(2)	LDC b	<table><tr><td>b 的地址</td><td></td><td></td></tr><tr><td>a 的地址</td><td></td><td></td></tr></table>	b 的地址			a 的地址			(无代码)			
b 的地址												
a 的地址												
(3)	LD	<table><tr><td>b 的值</td><td></td><td></td></tr><tr><td>a 的地址</td><td></td><td></td></tr></table>	b 的值			a 的地址			(无代码)			
b 的值												
a 的地址												
(4)	LDC a	<table><tr><td>a 的地址</td><td></td><td></td></tr><tr><td>b 的值</td><td></td><td></td></tr><tr><td>a 的地址</td><td></td><td></td></tr></table>	a 的地址			b 的值			a 的地址			(无代码)
a 的地址												
b 的值												
a 的地址												
(5)	LD	<table><tr><td>a 的值</td><td></td><td></td></tr><tr><td>b 的值</td><td></td><td></td></tr><tr><td>a 的地址</td><td></td><td></td></tr></table>	a 的值			b 的值			a 的地址			(无代码)
a 的值												
b 的值												
a 的地址												
(6)	ADD	<table><tr><td>b 的值</td><td>+</td><td>a 的值</td></tr><tr><td>a 的地址</td><td></td><td></td></tr></table>	b 的值	+	a 的值	a 的地址			(无代码)			
b 的值	+	a 的值										
a 的地址												
(7)	ST	<table><tr><td>a 的值</td><td>+</td><td>b 的值</td></tr><tr><td>a 的地址</td><td></td><td></td></tr></table>	a 的值	+	b 的值	a 的地址			(规范化)			
a 的值	+	b 的值										
a 的地址												
(8)	ST	空栈	ADD a, b									

图 9-3 虚拟栈中从内存到内存的代码

当一个存储操作发现待处理的目标与表达式元素中的一个操作数相匹配时，将生成一个基于内存的算术运算进行计算，而不是装入一个寄存器的值。任何时候只要表达式超过一个运算符，就会像先前那样分配一个寄存器，并生成必要的代码将它约简为一个运算符。原则上，这一类设计的复杂度极限就是目标机器体系结构的复杂度。如果计算机的寄存器到内存寻址模式仅支持加法和减法，编译程序就不必将乘法运算符保留在虚拟栈中。如果计算机有一些怪异的指令可在内存中执行乘法和加法的组合，编译程序就值得在单个虚拟栈单元中同时保存两个运算符和全部三个操作数，从而在参数使得有可能使用该指令时，这些信息都是可用的。

9.3.6 分支寻址选择

在编译程序设计中，通常会更关注分支寻址，因为程序员几乎无法控制这些寻址模式，并且在现代计算机体系结构的典型程序中，分支的范围很可能涉及多种寻址模式。此外，一条分支指令通常横跨多个分支，后面的寻址模式选择决策影响到前面决策的跨度的情况并不少见。

在一些计算机中，用于测试比较结果的条件分支仅使用短寻址模式，而无条件分支则既可以是短的也可以是长的。在这种情况下，编译程序必须定义一种“超长的”条件分支宏，其组成是一个条件分支（保留原条件）围绕一条无条件长分支形成的序列。然后，这个宏可看作一条非常长的指令，用于代码生成目的。

当已知一个目标地址远比长（或短）地址空间的边界更近（或更远）时，分支寻址第一次尝试即可正确地给出决策。在向后分支的情况下这特别容易实现，因为目标的物理地址是已知

的。向前分支则需要先执行一遍程序统计的数据流分析以提供信息，这一统计分析将计算可能的指令字数目，为后续优化工作留下很大的空间。剩余的分支通常被指定为缺省模式，并在后续分析中不断修正。

最优算法的复杂性与分支数目的平方成正比。它在所有无法确定的情况下产生最少的分支，然后审查每一分支的代码，将那些初步决策证明无效的地方替换为长分支。这可能导致其他分支无效，因而该过程必须迭代执行，直至最后稳定。注意，每次对任一分支寻址模式的修改都必须调整所有的分支，因为跟在被修改分支指令后面的代码将被移动以适应这一修改，任何跳转到被移动代码中的分支可能不再是正确的；被移动代码中的绝对分支、或跳出被移动代码的相对分支也变得无效。通常不推荐使用这一算法，因为可找到一个线性时间的算法，不必移动或删除生成的代码即可给出安全的决策；仅在非常罕见的情况下（通常是5%或更少），分支是次优的，这取决于代码中分支的密度以及短寻址模式的跨度。

线性算法需要使用一个队列数据结构，将与短寻址分支的跨度一样多的字放入队列中，队列中的每一个字可保存一个最小的指令字。代码生成程序将所有生成的代码输出到队列的一端，而不是直接将生成的代码写到输出文件中或已完成的代码块中；仅当队列为满时，才将代码从队列的另一端发送到一个输出文件中。

每一个未知的分支都生成一个跳转到队列中的长分支，所有分支在队列另一端出队时将被审查。如果一个向后分支入队时其目标仍在队列中，则已知该分支是一个短分支，否则为它生成一个长分支。一个向前分支在入队时先被假定为长分支（这不需要提前进行分析），如果在退出前目标地址已入队则修改为短分支。在代码入队时，可基于如下假设初步推断分支的偏移量：队列中的所有字是已完成的代码；当分支出队时计算实际的地址值。

这里假设目标地址的标记在队列中不占空间，并且分支与目标地址的标记一一匹配（跳转到单个地址的多个分支分别有多个对应匹配的地址标记）。在实际应用中，可为地址标记在队列中分配附加的字。类似地，如果每次一个目标地址入队时与之匹配的分支仍在队列中，则可用地址大小之差在逻辑上压缩队列；如果数据结构大得足以每一逻辑上的压缩容纳额外的代码字，则在物理上没有必要移动队列中的代码。可用一个单独的变量为实际的代码字计数，因而逻辑队列的大小仍是一个常量。注意每一分支出队时，字计数器可通过该分支的逻辑大小进行调整，从而记录其流量。代码清单 9.6 给出了一个分支寻址队列的代码。

代码清单 9.6 一个分支寻址选择队列（声明）

```
MODULE BranchQueue;
```

```
(* 用于生成相当好的跳转地址。用法:
```

```
  向前跳转:
```

```
  向后跳转:
```

```
-----
lab := UniqueLabel();
EmitJump(TRUE, lab);
EmitCode(theOp, opnd, nWords);
...
EmitLabel(TRUE, lab);
```

```
-----
lab := UniqueLabel();
EmitLabel(FALSE, lab);
EmitCode(theOp, opnd, nWords);
...
EmitJump(FALSE, lab);
```

```
然后在编译结束时调用 FlushQueue()。
```

```
*)
```



```

FROM TargetGenerator IMPORT
    CurrentAddress, (* 下一输出代码字的逻辑地址 *)
    EmitToTarget, (* 输出一条目标机器指令 *)
    BackPatch; (* 回填先前输出的短分支或长分支 *)

FROM CodeConstants IMPORT
    ShortJumpRange, (* 从下一位置开始的最远的短跳转 *)
    ShortJumpSize, LongJumpSize, (* 生成的指令字的数目 *)
    GenericOpcode, (* 操作码列表, 包括 ShortBr 和 LongBr *)
    AddressMode; (* 寻址模式列表, 包括 shrt 和 long *)

EXPORT
    EmitJump, EmitLabel, EmitCode, (* 将它们输出到队列中 *)
    UniqueLabel, (* 获取一个新标号 *)
    FlushQueue; (* 编译结束时调用该过程 *)

CONST
    maxtable = 500;

TYPE
    queueRange = [0 .. maxtable];
    queuedata = (other, justcode, jumped, fordjmp, backjmp
        , alabel, backlab, longlab, shortlab, deleted);

VAR
    QueueTake, QueueInto, QueueSize: queueRange;
    nextLabel: INTEGER;
    CodeQueue: ARRAY queueRange OF RECORD
        kind: queuedata;
        theOpcode: GenericOpcode;
        aNumber, theData: INTEGER
    END;

PROCEDURE FindLabel(theLabel: INTEGER): queueRange;
VAR
    index: queueRange;
BEGIN
    index := QueueInto;
    WHILE index <> QueueTake DO
        IF index = 0 THEN
            index := maxtable
        ELSE
            DEC(index)
        END;
        WITH CodeQueue[index] DO
            IF (kind > other) AND (aNumber = theLabel) THEN
                RETURN index
            END
        END (* WITH *)
    END (* WHILE *)
END FindLabel;

PROCEDURE UpdateQueue(addwords: INTEGER; additem: BOOLEAN);
VAR

```

```

recycle: BOOLEAN;
locn: INTEGER;
BEGIN
  IF additem THEN
    IF QueueInto = maxtable THEN
      QueueInto := 0
    ELSE
      INC(QueueInto)
    END;
    QueueSize := QueueSize + addwords
  END;
  WHILE (QueueInto # QueueTake) AND (NOT additem OR (QueueSize > ShortJumpRange)
  OR ((QueueInto + maxtable - QueueTake) MOD maxtable < 5)) DO
    recycle := FALSE;
    WITH CodeQueue[QueueTake] DO
      CASE kind OF
        other:
          EmitToTarget(theOpcode, theData, aNumber);
          QueueSize := QueueSize - aNumber |
        jumped, backlab:
          recycle := TRUE |
        fordjmp:
          IF theOpcode = ShortBr THEN
            EmitToTarget(ShortBr, 0, ShortJumpSize);
            QueueSize := QueueSize - ShortJumpSize
          ELSE
            EmitToTarget(LongBr, 0, LongJumpSize);
            QueueSize := QueueSize - LongJumpSize
          END;
          kind := jumped;
          theData := CurrentAddress;
          recycle := TRUE |
        backjmp:
          WITH CodeQueue[FindLabel(aNumber)] DO
            locn := theData;
            kind := deleted
          END; (* 内层 WITH *)
          IF CurrentAddress - locn < ShortJumpRange THEN
            EmitToTarget(ShortBr, locn, ShortJumpSize)
          ELSE
            EmitToTarget(LongBr, locn, LongJumpSize)
          END |
        alabel:
          kind := backlab;
          theData := CurrentAddress;
          recycle := TRUE |
        longlab, shortlab:
          WITH CodeQueue[FindLabel(aNumber)] DO
            locn := theData;
            kind := deleted
          END; (* 内层 WITH *)
          IF kind = shortlab THEN
            BackPatch(ShortBr, locn, CurrentAddress)
          ELSE
            BackPatch(LongBr, locn, CurrentAddress)

```

```

        END |
        deleted:
        (* 这已经被使用, 忽略它 *)
    END (* CASE *)
END; (* WITH *)
IF recycle THEN
    CodeQueue[QueueInto] := CodeQueue[QueueTake];
    IF QueueInto = maxtable THEN
        QueueInto := 0
    ELSE
        INC(QueueInto)
    END
END;
IF QueueTake = maxtable THEN
    QueueTake := 0
ELSE
    INC(QueueTake)
END
END (* WHILE *)
END UpdateQueue;

PROCEDURE EmitCode(theOp: GenericOpcode; operand, nWords: INTEGER);
BEGIN
    WITH CodeQueue[QueueInto] DO
        kind := justcode;
        aNumber := nWords;
        theOpcode := theOp;
        theData := operand
    END;
    UpdateQueue(nWords, TRUE)
END EmitCode;

PROCEDURE EmitLabel(forward: BOOLEAN; theLabel: INTEGER);
VAR
    mykind: queuedata;
    addwords: INTEGER;
BEGIN
    addwords := 0;
    IF forward THEN
        WITH CodeQueue[FindLabel(theLabel)] DO
            IF kind = fordjmp THEN
                mykind := shortlab;
                theOpcode := ShortBr;
                addwords := ShortJumpSize - LongJumpSize
            ELSE
                mykind := longlab
            END
        END (* WITH *)
    ELSE
        mykind := alabel
    END;
    WITH CodeQueue[QueueInto] DO
        kind := mykind;
        theOpcode := BackP;
        aNumber := theLabel;
    
```

```

        theData := 0
    END;
    UpdateQueue(addwords, TRUE)
END EmitLabel;

PROCEDURE EmitJump(forward: BOOLEAN; toLabel: INTEGER);
VAR
    mykind: queuedata;
    addwords: INTEGER;
BEGIN
    addwords := LongJumpSize;
    mykind := backjmp;
    IF forward THEN
        mykind := fordjmp
    ELSE
        WITH CodeQueue[FindLabel(toLabel)] DO
            IF kind = alabel THEN
                addwords := ShortJumpSize
            END
        END (* WITH *)
    END; (* IF *)
    WITH CodeQueue[QueueInto] DO
        kind := mykind;
        IF addwords = ShortJumpSize THEN
            theOpcode := ShortBr
        ELSE
            theOpcode := LongBr
        END;
        aNumber := toLabel;
        theData := 0
    END;
    UpdateQueue(addwords, TRUE)
END EmitJump;

PROCEDURE FlushQueue();
BEGIN
    UpdateQueue(0, FALSE)
END FlushQueue;

PROCEDURE UniqueLabel(): INTEGER;
BEGIN
    INC(nextLabel);
    RETURN nextLabel
END UniqueLabel;

BEGIN
    QueueTake := 0;
    QueueInto := 0;
    QueueSize := 0;
    nextLabel := 0
END BranchQueue.

```

(* BranchQueue 的初始化代码 *)

9.3.7 分支链

在同时支持长分支与短分支寻址模式的计算机中，以速度开销换取空间优化的一种更有趣的方法，是尝试以一个短分支跳转到具有相同终结目标的其他分支，从而取代原来的长分支。一个源程序中连续的 **ELSIF** 子句往往会产生一条无条件跳转指令，从每一子句的最后跳转到

IF 结构的最后。这些跳转指令的最后一条或两条很可能落在短分支的范围，但其中的第一条则未必。然而，如果每一条单独的子句不是特别长，则每一出口的跳转在短分支寻址范围内即可到达下一子句的出口跳转。因而替换更近的目标地址会导致多个短跳转的执行，而不是一个长的跳转。这通常实现为已生成代码上的窥孔优化，但也很容易修改树平展代码生成程序，可能基于分析程序或一遍统计数据流分析提供的一些信息，在生成代码时将分支组成一条链。

如果不考虑分支链，代码生成程序往往在到达 **ELSE**（或 **ELSIF**）时生成一条跳转指令，并且在递归调用生成该子句代码的 **StatementList** 返回时输出匹配的标号。如果将标号发送给生成 **ELSIF** 子句代码的非终结符，则可将分支组成一条链，从而恰好在一个分支生成自己的出口跳转指令之前可输出该标号。就算一个分支没有自己的跳转指令，它也必须输出这一标号，这增加了一些复杂性。代码清单 9.7 演示了这种优化的通用形式。类似的逻辑还可用于汇合一个 **CASE** 结构的所有支路的执行路径；再稍加修改，同一优化技术还可用于处理嵌套的 **IF** 语句。

代码清单 9.7 在一个代码生成程序的文法中构建分支链

```
Statement
→ "IF" IfStatement ↓0           { 尚无分支需要串成链 }
→ ...                           { 其他语句 }

IfStatement ↓chained:int
→ BoolExpn
   [UniqueLabel ↑myelse; EmitJump ↓myelse]
  "THEN" StatementList
   ([chained # 0; EmitLabel ↓chained])?
                                     { 此处将传入的分支串成链 }
(
  "ELSIF"
    [UniqueLabel ↑myexit; EmitJump ↓myexit; EmitLabel ↓myelse]
    IfStatement ↓myexit           { 向下传递自己的出口标号 }
  |
  "ELSE"
    [UniqueLabel ↑myexit; EmitJump ↓myexit; EmitLabel ↓myelse]
    StatementList
    "END"
    [EmitLabel ↓myexit]
  |
  "END"
    [EmitLabel ↓myelse]
);
```

从时 / 空折中这个统一体的另一角度看，自然链接在一起的分支也可由代码生成程序解开。嵌套 **IF** 语句可产生自然的分支链，其中内层 **IF** 的 **THEN** 子句跳过其 **ELSE** 子句后，直接又跳过外层 **IF** 语句的 **ELSE** 子句。若要消除第二条跳转指令，则需要合成一个出口标号的列表沿分析树向上传递，并在待生成的下一代代码不是跳转指令时输出。

9.4 代码生成的复杂性

代码生成本质上有两种基本上相反的方法，一种更简单，而另一种更优。简单的方法是在目标机器中尽可能地为抽象虚拟机建模。另一种方法寻求以目标机器代码对不同的抽象算法进行编码，然后尝试在中间代码树中识别该抽象算法的成分，从而可生成该算法的最优代码序列。

一个实用的代码生成程序往往采纳这两个极端的某一折中方案。本书已展示了寄存器分配情况下这种折中方案的一些选择范围。在寄存器栈的实现中精密地建模了抽象虚拟机，而在虚拟栈结构中携带运算符和两个操作数使得生成的代码几乎接近于专业级汇编程序员可能写出的最佳代码。

很重要的一点是应意识到，即便对于一个编译程序而言，计算机只是一个细微的考虑因素，但每一计算机都有图灵机的能力，即它有能力计算任何可计算的问题（在内存大小限制之内，包括联机文件存储）。我们在这里考虑的问题不是“可编译什么？”（该问题的答案是“所有东西”），而是“编译它的最佳方法是什么？”。如果一个商用编译程序无法为一个常见的程序序列找出最佳的代码，那么它在市场上就会被这方面的成功者取代。

像本书这类关于编译程序设计的一般性论著很难给出最优代码生成的特定细节，因为每一种计算机的指令集是不同的，这些差别往往微妙但影响深远。对这一问题的处理几乎总是需要划分为两个阶段。第一阶段要求采用机器语言或汇编语言为目标机器编写代码，以及评审其他专业级程序员编写的机器语言代码，从而对机器的指令集越来越熟悉。只有基于这样的洞察力，才有可能实现一个可输出相当不错代码的代码生成程序。第二阶段要求研究从各种样例程序生成的代码，若有可能则手工重写这些代码以改进其性能，然后再修改代码生成程序（通常是添加一些特殊处理的情况）以自动地生成改进的代码。第二阶段可以迭代任意多次，但不应省略这一阶段，因为再细心设计的编译程序在投入真正应用中处理真实程序时，也会产生一些令人意想不到的事情。

某一特定算法的最优代码序列可能并不是显而易见的。尽管大多数机器指令的设计都设想了其特定用途，但其中许多指令存在一些微妙的副作用，聪明的编译程序可加以发掘以取得性能上的改进。一个集成了尽可能多的已知优化技术的编译程序在生成每一个可能的指令序列时，都将寻求最少的代码以达到某种已报导过的特定效果，从而产生一些程序员手工从未考虑过的奇妙组合。

在接下来的章节中，本书将首先讨论比较正规的指令集上的代码生成通用问题，然后讨论一些处理怪异或不常见指令的例子。

9.4.1 指令选择

任何真正的计算机语言，包括 Modula-2 和 COBOL 等高级语言以及作为本章主题的低级机器语言，均提供了四则算术运算、将值赋给一个变量或从变量恢复当前值、比较两个值并根据比较的结果确定控制决策、跳转到封闭例程并从中返回等机制。大多数语言还支持多种数据类型或数据大小，并约定了它们之间相互转换的规则。一些语言自动地将不同的类型强制转换为兼容的形式，而另一些语言（如 Modula-2 语言和大多数机器语言）则要求显式地转换。

并非所有的计算机语言都支持所有令人感兴趣的数据类型。例如，沃思设计的 Modula-2 语言缺少一种复杂算术运算的数据类型；许多小型的计算机没有浮点运算的硬件指令，有些甚至未提供整数除法指令。每一种语言的图灵机计算能力保证了所有缺失的特性能够从可用的特性集构造（模拟）出来，这正是代码生成的重要组成部分之一。如果一个编译程序的源语言支持整数除法，而其目标机器却没有这一特性，则该编译程序必须生成一个合适的指令序列以软件方式实现除法，常见方法是调用一个由编译程序提供的库例程。因而，编译程序提供的库例程（或编译程序可用的其他方式）扩展了目标机器的指令集，编译程序的真正目标机器是如此扩展后的虚拟机。

分析程序和约束程序有义务找出被编译的源语言中每一运算符的准确操作和数据类型。这些信息可直接传递给代码生成程序，也可附加在中间代码树上作为结点和装饰。虚拟目标机器语言中必须至少有一种运算符与中间代码中的每一抽象操作和数据类型相对应，惟一困难的工作就是在有多个运算符时如何从候选清单中作出抉择。

一个树遍历文法可寻找特定的树模式，并代之以标记了使用特定指令或指令序列的结点，从而完成一些指令选择工作。如果目标机器上的乘法运算比移位运算慢（通常如此），则一棵表示与 2 的常量幂相乘的子树模板可转换为一个左移位运算。如果乘法与移位或加法有较大的性能差距，那么一个涉及两个 2 的幂之和（或差）的乘积（譬如 3、5、7、9、10、15、20、30、40 等）也可考虑这类优化。然而应注意，任何有意义的副作用必须予以保留。例如，如果一种语言要求报告算术运算的溢出信息，则生成的代码应检查从结果的左端移出的位，并在出现超范围乘积时执行相同的动作。此外还应注意的是，第 8 章所述的诸如消除范围检查等数据流分析可令许多这类溢出测试变得没有必要。

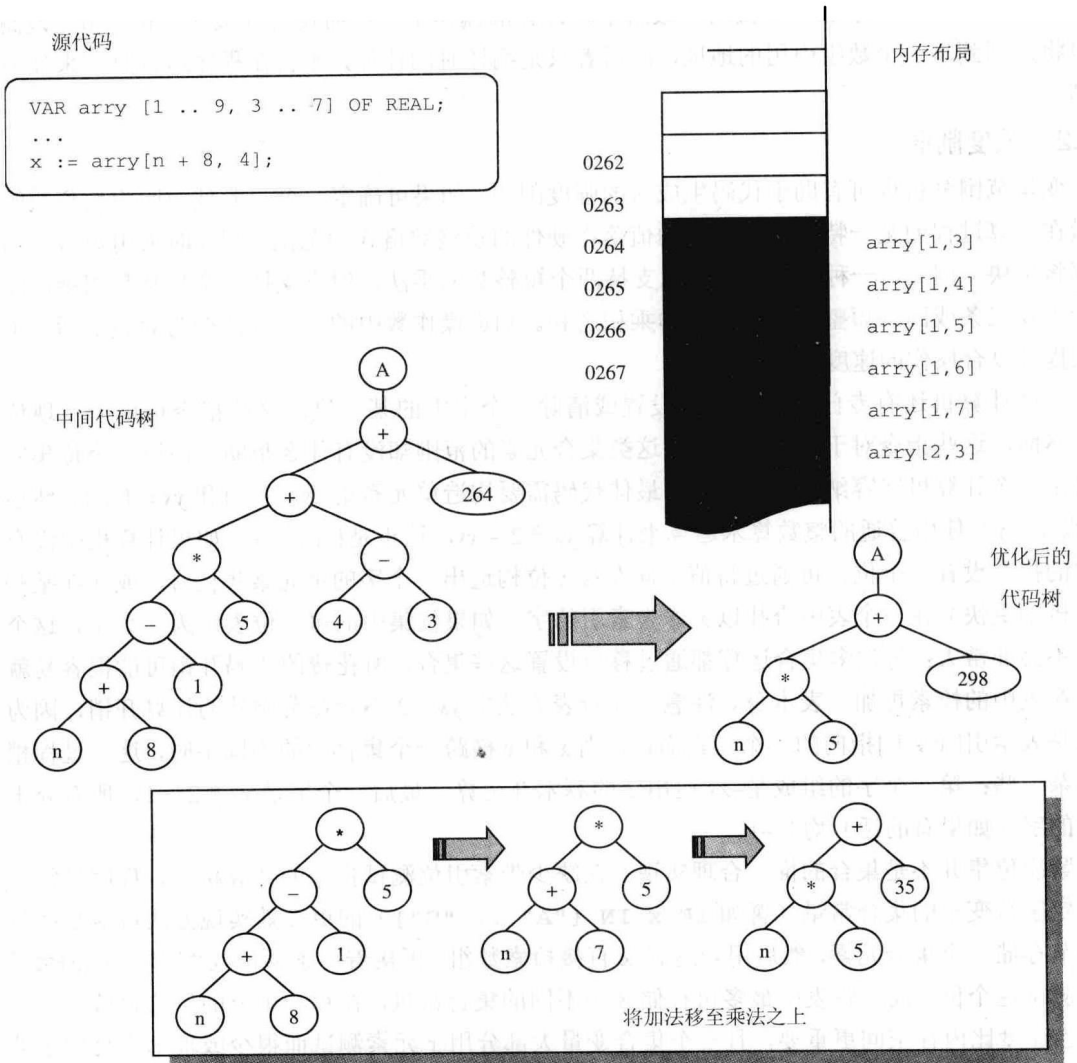


图 9-4 在数组下标计算中向外传播常量。其中的插图展示了加法如何移出到乘法之上

一个特别有意思的优化是识别一个记录字段的选择中出现的常量偏移量合计，或栈中（或全局帧中）常量数组下标和变量偏移量的合计。当数据下标范围的下界不为0时，即便数组下标不是一个常量，也可将地址中一些常量成分向外传播给变量引用，以减少运行时所需的算术运算。注意，为避免运行时不必要地计算下标基址偏移量，必须对树进行变换，从而将一个常量与涉及另一常量的和（或差）的乘积转换为一个乘积之和，如图9-4所示。然后在运行时通过一个显式的运算符完成乘法，加法则与隐式的和合并到硬件的变量引用中。如果寻址操作足够明显，在遍历代码树的常量折叠优化中即可实现这一变换 [Pittman, 1985]；如果虚拟栈以本章前述方式容纳了一个运算符和两个操作数，也可在寄存器代码生成的模拟执行中实现这一变换。两种方案的结果是相同的：单个变量引用以及一个计算得到的偏移量，偏移量从符号表中定义的偏移量开始计算。

在某些计算机上，翻译数组下标地址的引用还可改进 **FOR** 循环的控制变量的测试。如果硬件支持0测试或符号测试的代码少于与另外的任意值进行比较的代码，则编译程序就可考虑为 **FOR** 循环的控制变量添加一个仅用于数组下标计算的偏移量，从而使得其最终的值为0。该偏移量将加回到每一个数组引用的地址，而后者只是编译时的计算，不会在程序运行时带来任何开销。

9.4.2 强度削弱

变量范围分析也可有助于代码生成时的强度削弱；如果可确定一个一般被声明为长整数的变量在计算过程的某一特定点所包含的值落在硬件的短整数格式的范围，则此时采用短格式的计算将更快。例如，一种流行的计算机支持两个短整数的乘法，但不支持两个长整数相乘；而后者需要三条或四条短整数乘法指令的乘积之和。知道操作数中的一个或两个为短整数后，可大大提升复合操作的速度。

一些计算机还有专门的指令用于设置或清除一个字中的某一位，这些指令可用于实现位集。然而，这些指令对于生成 $\{x..y\}$ 这类集合元素的范围却没有什么帮助。假设一个位集完全可由一个计算机字容纳，这类计算的最佳代码需要构造单元素集 $xs = \{x\}$ 和 $ys = \{y\}$ ，然后（假设 $x \leq y$ ）使用普通的整数算术运算来计算 $ys * 2 - xs$ ，这正是 $\{x..y\}$ 。如果计算机中没有专门的指令设置一个位，可通过将值1向左移 x 位构造出一个字的单元素集 $\{x\}$ ，或（在某些计算机中更快）在一个表中查找以 x 作为索引的字。如果位集中的每一位表示为一个字，这个表并不会非常大；每次多集合运算都通过移位设置这些集合，所花费的代码开销可能很容易就超过在表中的检索再加上表本身。注意，在查表方法中 $ys * 2$ 不会花费额外的计算开销，因为它只是表索引 $[y+1]$ 指向的一个元素而已。当 x 和 y 横跨一个集合中的不同字时，这一过程稍微复杂一些；第一个字的组成是 $-xs$ 运用2的算术补运算，最后一个字是 $ys * 2 - 1$ ，所有介于中间的字（如果有的话）均为-1。

紧凑位集并不是集合的惟一合理实现。在缺少带索引位测试指令的计算机中，程序员常用于限定字符变量的集合常量（譬如 **IF x IN {"A" .. "Z"}**）的更高效实现方式可能是在每一字节存储一个集合元素，然后用表达式 x 直接检索数组，再接着一条移位或“逻辑与（**AND**）”指令测试这个位。同一张表中最多可存储8个不同的集合常量，在每一个位的位置存放一个常量。当速度比内存空间更重要，且一个集合变量大部分用于元素测试而很少或根本不会用于涉及并集或交集的表达式中时，也可采用相同的方式实现集合变量，每字节表示一个元素。当然，

你不能像集合常量那样压缩多个集合变量，但元素测试的速度更快了，因为不需要任何移位或掩码运算。至于在编译时是否不压缩一个集合变量，最好依据对程序统计分析后提供的信息来作决策。

即便只有一个非常有限的机器指令集，生成创造性代码的机会仍是无穷的。如果一个编译程序的目标机器仅有很小的内存，编译程序可用一条需要双字节地址的内存引用指令替换一条跳过后两个字节代码的无条件分支指令：内存引用指令是一个比较运算，将该双字节作为地址，从由此寻址的任一随机内存位置取数，再执行比较，然后忽略结果并继续执行后续的指令；另一执行路径则跳过比较操作码，直达作为其地址部分的双字节的首字节，并从这里恢复执行。我们可能会批评在汇编语言中手工嵌入这类代码技巧的程序员，但编译程序这么做时可保证正确性和安全性，或在无效时拒绝这么做。

9.5 专用指令

计算机硬件设计人员似乎无法避免经常屈服于专业级汇编语言程序员索要这种或那种特殊指令的压力。一个简单的编译程序将直接忽略这些华而不实的怪异东西，切实奉行手工编写的汇编代码比编译程序生成的代码更为高效这一广为接受的信念。只要为代码生成程序付出更多的心血，细心的编译程序设计人员可采用适当方式利用这些神秘指令，使得开发出来的编译程序更有价值。在一个较快的编译程序中充分挖掘计算机资源的潜能，确实可算得上是一种艺术工作。

一种常见的微处理器有好几条指令被设计为迭代地执行，另有一条特殊的“迭代”指令可实现这一目标。这些指令的本意是支持字符串数据，但它们还可用于实现对记录或数组这类数据结构中的赋值。编译程序设计人员应仔细分析实现重复搬移的代码，当数据结构不长于一个字时，选择由一个个单字装入和存储指令组成的序列，因为循环花费的执行时间开销通常大于对应的展开后的装入和存储操作。

另一种常见的计算机缺少迭代指令或字符串指令，但支持多个寄存器的装入和存储。这两条指令的本意是在一个中断服务例程中保存和恢复处理器的状态，以及在一个过程的入口处分配非易变的寄存器。它们也可有效地用于搬移大于一个或两个字的数据结构。如果一个数据结构远远大于可用寄存器所能容纳的数目，通常最有利的做法是将尽可能多的寄存器溢出到内存中，然后将它们全部用于一个长搬移循环中的多个装入和存储操作。于是循环的开销有效地分摊到所有相关的寄存器，而不是在搬移每一个字时都付出所有的开销。

只要对循环进行充分的分析，就可将同一技术用于展开源代码中用一个 **FOR** 循环编写的数组初始化代码。存储到数组中每一元素的初始值可在若干寄存器之间复制，寄存器的数目划分了数组的大小；然后调整循环控制变量，按需要执行更少次数的循环。如果不可能找到数组大小的一个精确约数，则选择一个小于整个数组的最方便的大小，最后再在循环之外填充剩余的字或字节。如果待存储的值不统一（非 0），也有必要选择若干寄存器恰好涵盖元素大小的倍数，或为该循环添加单独的存储指令，以保持这些块的大小一致。

9.5.1 RISC 和流水线处理器调度

尽管并行化和流水线算不上是计算机体系结构的最新成果，但它们在计算机设计中已越来越流行。一个计算机流水线将它执行一条指令所耗费的时间划分为离散的时间段，并在硬件中

的顺序部件中执行它们。当指令的执行前进到下一阶段时，每一部件被释放并开始处理下一指令。尽管单个操作并没有加快，但在同一时间有多个操作处于其执行过程中的某一阶段，从而产生了更大的总吞吐量。然而，不同于简单的计算机，这意味着一个运算的结果可能无法立即被下一运算使用。

例如，考虑以下程序片段：

```
a := x + y;  
b := z * 3;  
c := a * b;
```

第二条赋值语句原则上可在第一条语句完成之前开始执行，因为它不依赖于第一条语句的任何结果，况且其算术运算也可能要求使用不同的（可能是独立的）硬件。然而，第三条语句同时依赖于前两行的计算结果，并且依赖于相同的算术运算；在一个流水线计算机中，会要求该语句在执行其计算之前等待。通常流水线计算机的编译程序有义务安排程序的执行步骤，使得相互独立的运行可重叠执行，并且当序列处理所需的输入数据变得可用时，调度它们开始执行。另一些计算机体系结构提供了多个算术运算单元，也可取得类似的效果；此时编译程序也必须调度这些操作，使得在有需要时所需的算术运算单元也是可用的。在某些计算机中，硬件会在有需要时插入一些延时指令；但在另一些情况下，可能要求编译程序必须完成这一任务。

指令调度的最基本需求之一是流水线分支指令，其中分支指令的下一指令在该分支指令生效之前就已开始执行。这意味着编译程序必须将该指令安排为一条与分支指令没有依赖关系的指令，或插入一条或多条纯粹耗费时间的指令。

指令调度问题在不少方面类似于寄存器分配问题，但两者之间仍有很大的区别。指令调度的实现通常是在代码生成程序结束后的一次窥孔优化；除了个别的分析，指令调度并不能很好地遵循基于文法的树变换实现。此处描述的算法强调单个基本块的指令调度，并假设正在调度的基本块代码可生成到一个临时的数据结构中，其中的操作可按任意次序转换为最终的目标代码文件。类似于寄存器分配，最优算法可能要求采用图着色技术；本书的简化方法通常可对一大类流水线计算机硬件在线性时间内产生相当好的结果。

用于指令调度分析的数据结构是一个有向无环图（DAG），类似于第8章消除公共子表达式中的用法，其中的有向边表示一种计算依赖关系。每一操作（内存访问或算术运算）是图中的一个结点，并被连接到其他结点（表示依赖于这些结点的计算结果），如图9-5中首尾相连的方框所示。每一结点关联一个时间段（在示意图中表示为方框的长度），其依据是自该操作开始执行、直至计算结果可用的这段硬件处理时间。

一个完整的基本块的示意图将包含一个或多个独立的子图，这些子图主要由操作的依赖关系链组成。图9-5中有两个独立的子图；其中的大图汇合了几条不同的链，最后又将它分为两条链。开始构造该图时，结点（即方框）之间的排列表示尚未考虑指令调度时各个操作经过处理器流水线的执行时间，因而整个基本块的最长链的长度表示一条资源分配的“关键路径”。于是可以确定地生成调度后的目标代码：选择不必等待前一指令计算结果的最长链的第一个结点，从该链中删除这一结点，并在其位置插入适当的延时。只要处理器单元可用于启动新的独立计算，在下一指令周期中可启动另一处理，并总是选择不必等待未完成结果的最长剩余路径。

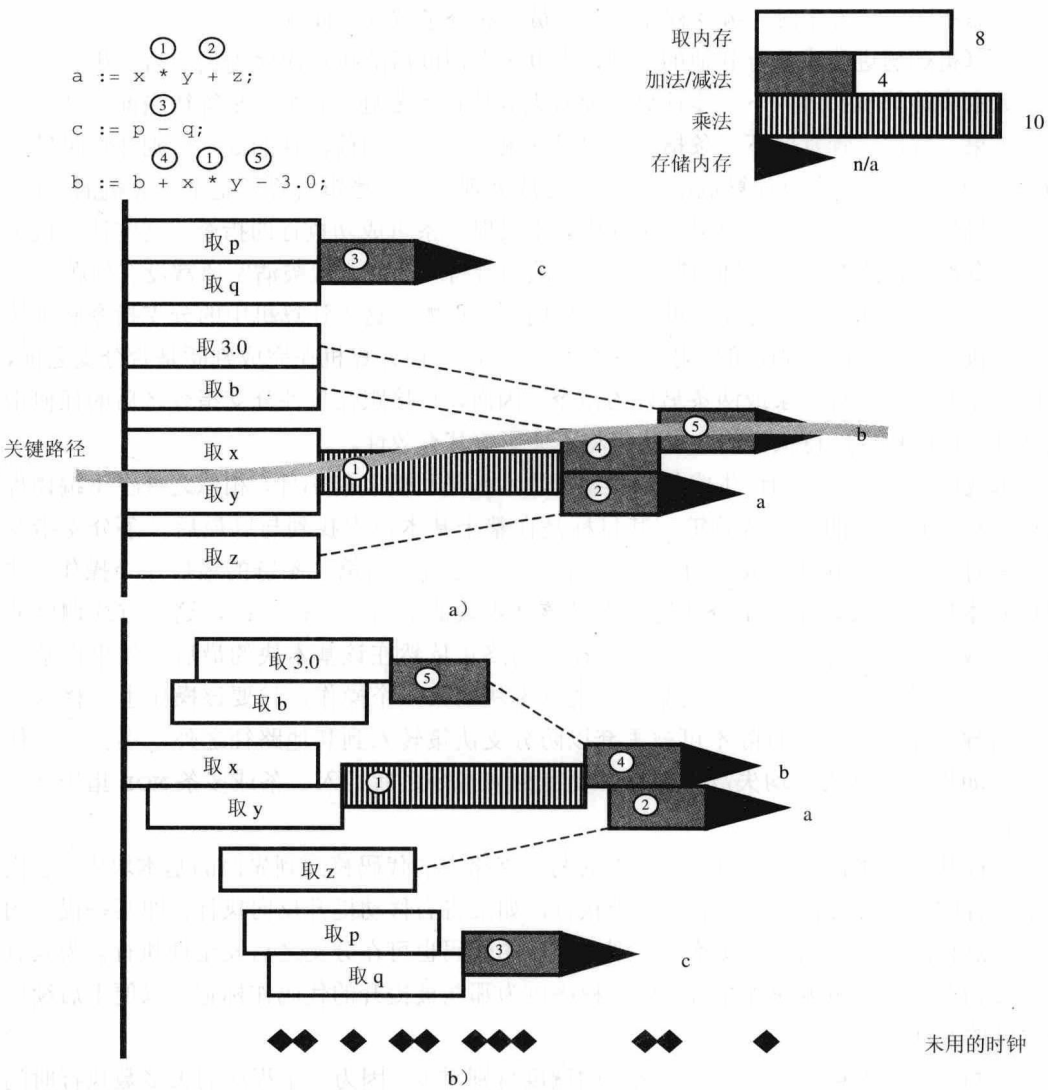


图 9-5 基于 DAG 关键路径的处理器调度。图 a 展示了最长的（关键）路径；图 b 是最优调度之后的情况

如果计算机有固定数目的几个不同处理单元均在完成其操作之前不可重新利用，这一简化的调度算法可能无法产生最优的调度。问题出在第二条链有机会占据了紧随其后的关键路径所需的资源。然而，该算法对于完全流水线的算术运算硬件而言是最优的，因为这种计算机总是准备好了启动任何新的操作。利用相邻基本块的少许信息，还可提升基本块边界处的算法性能。

资源调度的分析工作可在一个树文法中实现，该树文法构造了一个有向无环图并从中找出关键路径。树结点也可装饰调度信息，但以调度后的次序从树文法生成代码却是相当笨拙的，因为最优执行时间中的顺序操作可能分散在多棵子树中。

在图 9-5 的示例中，很明显的是将表达式树平衡后可令关键路径缩短。根据一般的表达式从左到右求值次序，在关键路径上原本有两个加法运算等待乘法运算的结果；运用运算符的结合律这一数学原理后，减法可与乘法运算并行执行，从而缩短了执行时间。然而注意在大多数计算机中，加法并不是严格地遵循结合律，并且在这两种求值次序中浮点运算的舍入效果会有

差别。在考虑不同候选方案时，编译程序设计人员应充分了解这一问题。

如果计算机还想进一步执行其他的处理，图 9-5 中调度后的执行次序提供了若干机会。许多并行和流水线计算机在进入下一步计算之前若无其他任务要处理，则只是等待当前活跃的处理单元产生结果；软件只管发射下一条指令，如果它依赖于一个当前活跃的处理单元，则计算机转入等待。在最新设计的许多高速计算机中，等待未完成处理的额外逻辑将降低整个计算机的速度。为这些计算机生成代码时，要求为每一时钟周期都发射一条可成功执行的指令。这些计算机中的大多数指令被设计为在单个时钟周期内执行，因而等待一个流水线被清空通常没有问题。

分支指令是一个例外。为与计算机其余部件的速度匹配，这类计算机中的分支指令必须按流水线方式执行。这意味着程序可发射一条条件分支指令，但计算机在完成判断是否分支之前，将继续取指令并开始执行一条或两条另外的指令。因而，直接跟在条件分支指令之后的任何指令必须具有这样的特性：待执行的分支决策将不会影响其有效性。

为流水线的分支指令正确地生成代码类似于流水线结构的功能部件，相似之处在于编译程序必须找出整个基本块的执行依赖链。其目标是在整个基本块中找到与以最后一条分支指令结束的链并行的一条依赖链（或链的片段），并在分支之后输出第二条链的最后一个操作。注意，根据基本块的定义，分支指令在逻辑上是该基本块的最后一个操作；以这种方式调度分支指令并不改变逻辑操作的次序，因为分支操作的终止依然在该基本块的最后。如果在基本块的结束处没有并行的链，则可利用某一后继基本块的第一个操作，只要该操作也可移入所有其他的前导基本块中，并且除不可被丢弃以防分支决策转入到其他路径之外，不会产生任何副作用。如果这两种尝试均失败，则有必要在分支指令之后插入一条或多条 **NOP** 指令（表示无操作）。

左移动提升这种优化技术已将一个分叉的两个支路中的代码移回到先前的基本块中，它们总是位于并行的链中，可在分支之后安全地执行。如果将右移动提升反向执行，即某些提升的代码从汇合点移回到分叉的两个支路中，则提升后的代码也可在分支之后安全地执行。为具有流水线分支指令的计算机编译程序时，优化程序可为那些被提升的代码作标记，以便于后续分支指令调度的实现。

那些返回到一个循环开始处的分支指令的调度特别重要，因为一个程序的大多数执行时间都花费在内循环。如果在循环的最后一个基本块找不到一条并行链的指令，则应移动循环开始处的第一条指令，既要移至循环的结束处，也要复制到循环开始之前的初始化代码中。当然，这里假设了第一条指令的结果在循环退出时可被丢弃，或循环最后的分支是一个无条件分支。图 9-6 演示了循环分支指令调度的各种不同情况。

9.5.2 向量处理器

计算机设计人员为取得更快的执行速度所采用的另一技术是扩展并行性，使之涵盖在一个循环中对一个数组（即向量）的所有元素执行本质上相同的处理。向量处理器的硬件设计是将单个或少量指令应用到一系列的值，这些值要么规则地分布在内存中，要么预先装入到向量寄存器中；然后在另一内存数组或向量寄存器中产生结果。另一种变形是有多个并行算术处理器，全部同一指令序列控制，每一处理器均在整个操作数数组的一个切片上操作。由于这些操作也是以流水线方式执行的，向量处理器通常可在每一时钟周期重启一个新数组元素上的计算，有时产生的吞吐量超过了计算机的时钟频率。

```
n := x;  
REPEAT  
  a := a * b;  
  n := n - k  
UNTIL n < 0;  
(* 此时 n 不再活跃 *)
```

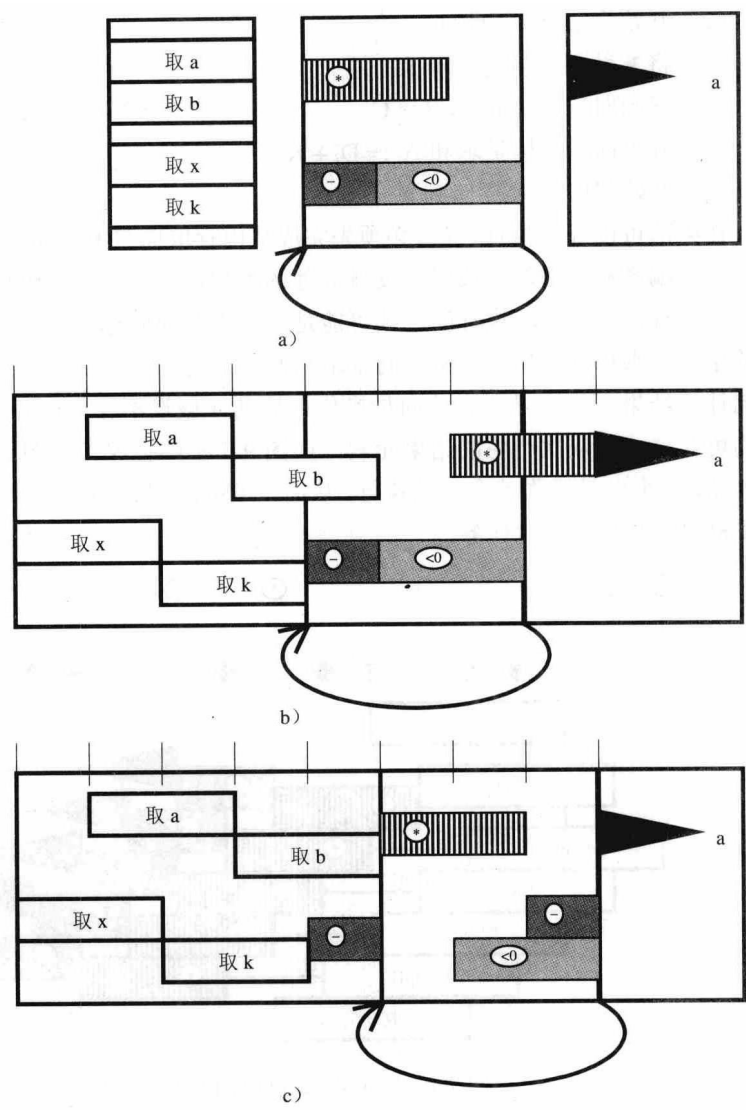


图 9-6 流水线循环分支指令的调度。图 a 表示循环不变量被移出后的未调度情况；图 b 表示在分支之后调度第二条链的最后一个操作；图 c 表示将操作从循环前面移至分支指令之后以及初始化块之中

采用这些技术的科学计算专用计算机的性能按 megaflops 或 Mflops 度量，这一单位表示每秒百万次浮点运算（Million Floating-point Operations per Second）。对于以下代码片段所示的循环（其中除变量 **i** 之外，所有变量均声明为 **REAL**），一台时钟周期时间为 10^{-8} 秒的向量处理器应能取得接近 200 Mflops 的平均处理速率：

```
FOR i := 1 TO 100 DO  
  a[i] := b[i] * c[i] + k  
END
```

向量化的编译程序将展开整个循环，代之以一个或两个向量操作序列（取决于处理器的向量长度）。尽管不同处理器的细节会有很大差异，这一操作序列基本上形如：

- (1) 开始将 **b** 装入向量 **B**

- (2) 开始将 c 装入向量 C
- (3) 将 k 装入标量寄存器 K
- (4) 开始向量乘法 $D_i := B_i * C_i$
- (5) 开始向量与标量求和 $A_i := D_i + K$
- (6) 开始将向量 A 存储到 a

在乘法可以开始执行之前,必须先完成从内存取向量 B 和 C 第一个元素的操作。如有可能,一个好的编译程序将在那段时间安排其他的处理;如果一个循环必须划分为在两个或多个向量片段上执行,那么循环执行的延续可能是一个主要的候选者。上例中,只有经过若干时钟后乘法的第一个乘积项产生了结果,向量求和操作才可以开始执行;并且在求和完成之前,不可以存储计算结果。但一旦所有的向量操作已启动(假设不存在内存计时的冲突),在每一时钟周期均可存储一个依次产生的结果元素,如图 9-7 中的抽象计时图所示。此外,无需程序的进一步关注,整个向量操作就会继续执行,从而只要处理器设施可用并且还有无关的操作需要执行,计算机可继续执行其他任务。

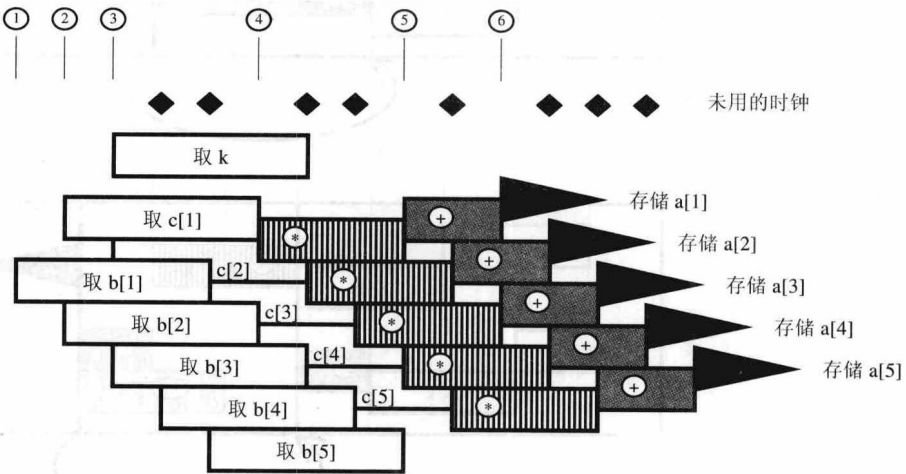


图 9-7 向量处理器对 $a[i] := b[i] * c[i] + k$ 的操作

编译向量代码同时涉及流水线处理器调度和循环展开这两个问题,并掺和了一些特殊情况的指令选择问题。例如,一个向量处理器有一个部件将一个标量累加到向量之积,且不会有额外的时间延迟,此时就不需要计算中间向量结果 D_i ,只需将整个由三个操作数组成的“积”与“和”直接定向到结果数组中。大多数向量处理器能够计算单个标量结果(称为约简变量),例如向量操作数的元素之和或之积,但关于如何求值的限制则每种机器会有不同。每一个向量可处理一个位于连续内存位置的数组中的元素;有些可接受大于 1 的跨距(即向量元素之间的地址增量),因而对于一种有更多限制的机器,编译程序必须以某一其他方式适应多维数组的按列(即非连续的)引用。即便有可能采用一个更大的跨距,如果跨距是内存槽数目的倍数,将地址分布到内存槽也可能导致冲突和性能下降。一个特别聪明的编译程序可能能够在较低端的下标范围插入一些哑元数组元素以避免这一问题。因而,如果程序员声明了一个变量的类型是 `ARRAY [1 .. k, 1 .. 64] OF REAL`,编译程序可能会留意到内层的下标范围是 8 个内存槽的倍数,并且好像范围为 `1 .. 65` 那样来编译它(在计算中直接忽略多出的元素)。如果

程序中不存在内循环从第一个下标开始遍历整个数组的所有下标,这一优化不会产生任何作用;并且如果存在对角引用(例如在含 $a[i, j]$ 的同一循环中有 $a[i - 1, j - 1]$),则实际上还会产生反作用。

向量化编译程序通常寻找机会将内层 **FOR** 循环转换为向量代码,只要循环中所有对循环出口处仍活跃的变量的赋值是由下标指定的数组元素或是约简变量,并且所有数组下标是循环不变的或线性的(按一个常数增量依次递增,且维数不超过一维)。即便这些条件都得以满足,仍有几个考虑因素会妨碍向量化,例如数组元素求值过程中的次序依赖关系。其中一种依赖关系称为“递归”(尽管这与过程的递归没有关系),指有一些数组元素在循环中计算,但在循环后续迭代的某些部分又再使用这些元素,例如 $s[i] := s[i - 1] * k$ 。在许多情况下,编译程序可用一个临时的标量持久循环变量取代在下一迭代中将使用的数组元素: $t := t * k$; $s[i] := t$; 但在更复杂的情况下,检测这样的机会需要更细致的数据流分析。当递归横跨多次迭代时,编译程序更加难以执行正确的动作。当数组下标由表达式求值,且其值在编译时无法知道的情况下,编译程序判断是否可能是递归的负担也加重。例如,在以下循环中:

```
FOR i := n TO m DO
    a[i] := a[i + j]
END
```

如果变量 j 的范围分析无法确定其值可否为负数,那么编译程序就不可假设该循环不是递归的。

许多循环用一条条件语句限定循环中的某些计算,该条件语句基于的计算结果要么来自循环之中,要么来自循环之外。条件语句分支无法向量化,但处理器架构设计人员早已凭借自己的创造能力,给出了可取得相同计算结果的可向量化方案。编译程序为实现同一目的,也有若干步骤须执行。本书之前已讨论过循环开关外提,作为循环不变条件测试的一种优化方法,它与这里讨论的问题关系密切。对循环控制变量的测试也可分解到循环之外。如果一个数组中有单个元素的值受保护,而所有其他元素的值均被循环中的计算结果取代,且受保护元素的下标是循环不变的,此时更简单的方法是在进入循环之前直接保存这个元素,然后在循环体无条件地替换了所有元素之后,再恢复该元素。类似地,将循环控制变量的范围测试划分为若干更短小的循环也是合算的,每一循环对应一个完整的子界,并且每一循环包含该子界的无条件代码。

在向量处理器中,处理条件语句的硬件支持通常以某种方式禁止将一个结果赋值给一个向量元素,只要另一向量中对应的值满足某一条件。控制向量可能是一个简单的位向量,其中的每一个位对应向量中的每一元素;它也可能是另一数据向量的符号或 0 测试。将循环中的一条条件语句正确地转换为向量化的条件赋值语句,需要仔细地分析各种不同的特殊情况,但所用分析技术并未超出本书讨论的技术,尽管稍微超出这里的论述范围。注意,只要能令更多的计算被向量化,编译程序设计人员应可自由地为中间结果分配一个向量临时变量(正如图 9-7 例子中的做法),即便在源代码中或按传统思维方式会想到使用一个标量值。由编译程序分配的向量临时变量,以及程序员声明的那些在循环出口处不活跃的数组,只需单独保存在计算机的向量寄存器中即是安全的,从来不用存储到主存中。除了初始化时间和单个向量化结果的传播延迟,链接到同一计算的额外向量操作本质上是无开销的,并且只会增加有效的 Mflops。

因而，在如下源程序的循环例子中：

```
FOR i := 1 TO n DO
  x := a[i] * b[i];
  IF x < 1.0 THEN
    r[i] := b[i] / c[i]
  ELSE
    r[i] := SIN(b[i]) - x
  END
END
```

一个谨慎的编译程序可能无条件地将表达式 $x[i] := a[i] * b[i]$ 、 $t[i] := b[i] / c[i]$ 和 $s[i] := \text{SIN}(b[i])$ ，以及复合值 $q[i] := x[i] - 1.0$ 和 $f[i] := \text{SIN}(b[i]) - x[i]$ 的求值结果保存到向量临时变量中。最后对向量 $r[i]$ 的赋值则可利用一条基于 $q[i]$ 符号的条件语句限制，在 $t[i]$ 和 $f[i]$ 之间作出选择。

注意，向量化的数学库函数一般返回一个完整的向量作为函数的值，其时间开销是花费在函数结果的求值时间，再加上向量中元素个数的小倍数（可能是每元素 1 个或 2 个时钟周期这样的小数目）。因而，即便该循环中只有相当低密度的不成立条件的求值，丢弃无用的三角结果在性能上仍将优于每次有需要时分别调用库函数获得单个标量结果。

嵌套的循环为向量处理器硬件提供了另一些优化的机会。当内层下标范围完全横跨一个多维数组时，该多维数组可以线性化，从而向量处理器硬件的每一向量配置都有更多数目的元素可以处理。然而，当复合的线性数组超过向量处理器硬件的长度时，这种做法的好处就会变小。当对跨距的考虑或求值次序依赖关系导致内循环的向量化不可行时，可尝试颠倒一对嵌套的循环，其结果有可能是新的内循环可被向量化。当外层控制变量的范围远远大于内循环的控制变量范围时，也可能希望反转一对嵌套的循环。

9.6 代码优化的变形

9.6.1 代码优化的分类

习惯上，将代码优化的考虑因素划分为两大类：一类是机器无关的优化，可在抽象语法树上执行而不必考虑目标机器硬件；另一类是机器有关的优化，需要十分熟悉目标机器，这些知识既要用于实现，甚至也要用于提出需求，并且还常常用于中间代码的低级四元式表示或最终的目标代码本身。在考虑常量折叠、循环不变代码移动这类明显的机器无关优化时，我们所持的观点往往不太常见，即大多数优化技术或多或少地依赖于目标机器硬件的体系结构，因而我们发现这一区别并没有太大作用。例如，尽管将整个循环不变语句移出循环之外几乎肯定是有好处的，但提炼出部分表达式加以移动的价值，则在很大程度上取决于保存和恢复中间结果与直接求值的开销相比较的相对开销。根据可用于保存中间结果值的寄存器数目，当无法使用寄存器时从内存重新装入中间结果值的额外时间，以及待处理的表达式的复杂程度，不同机器上的决策点可能会有相当大的不同。

参照本章与第 8 章的主要区别，可明显看出合理划分优化技术种类的另一尺度，即一些优化技术比另一些技术更容易采用一种文法驱动的实现。尽管按这一尺度划分会有很大的区别，但应指出的是，类似于机器有关或无关的优化，这里更像是一个连续体，而不是简单地一分为二的“有”或者“没有”。第 8 章讨论的右移动提升在树变换文法中相当笨拙，但并不是不可

能的；本章介绍的循环范围分析比右移动提升更容易实现为一个 TAG，但又不如常量折叠或消除公共子表达式那样容易。

组织优化技术分类的第三个尺度可能是它们实现起来比较合算的次序，或一条时间线。这同样只是更加有趣而已，而并不是更加实用，因为许多优化为其他优化提供了机会，反过来又可能将机会传回给最初的优化程序。例如，循环不变代码移动要求先执行常量表达式分析（常量折叠的一部分）；移出一大块代码后，循环体可能满足展开的条件，而展开又再次为常量折叠提供了机会。尽管如此，某些优化仍应在编译过程的早期发挥作用，而其他优化则有必要在后期执行。回代那些仅被调用一次的过程应尽早执行，因为这一变换的主要优势在于它与其他优化（譬如常量折叠和消除公共子表达式）开辟了机遇。另一方面，寄存器选择有必要作为最后的分析工作之一来执行，因为任何其他的优化都很容易导致其决策无效。

9.6.2 窥孔优化

一本编译原理教材如果不提及窥孔优化这种众所周知的机器有关优化技术，就说不上是一本完整的教材；尽管该术语实际上表示后期的各种代码修补，这些修补主要用于在代码生成阶段修正一些懒惰或倒霉的选择。经典的窥孔优化通常包括寄存器装入和存储之间的复写传播、算术运算符的强度削弱和内存访问的强度削弱（即寻址模式的选择）以及分支链接等。其中，如果实现了类似本书所述的基于文法的优化程序，则只有分支链接是可能有需要的。随着小型计算机的内存不断扩大以及对执行速度的不断强调，代码大小与执行速度之间的折中越来越倾向于执行速度而不是代码大小，因而除了最小的微控制器之外，其余计算机均消除了对窥孔优化的最后需求。

窥孔优化背后的思想是在代码生成程序完成其工作后，在生成的代码之上执行另一次遍历，可能能够找到机会进行一些小改进，其方法是每次仅观察一小段代码（正如从钥匙孔中窥视代码，由此得名）。例如，一个寄存器存储操作如果后接一个装入到相同寄存器的操作，则可修正为将冗余的装入操作消除；装入常量 0 可以代之以寄存器清除指令；诸如此类。我们通常推荐在代码生成之时执行这些替换，或甚至像树变换这样的更早时候，如此则没有什么必要回到代码文件来修正它们。此外，代码修正会占用更多的处理器时间，因为被改变的代码上的分支也需要作相应的修正。

小结

在编译的代码生成阶段，从优化程序接受的中间代码被转换为机器语言代码。代码生成非常依赖于机器，因而本章讨论了不同的目标机器体系结构，包括传统的计算机以及 RISC、流水线和向量处理器等。本章讨论了非常规的计算机体系结构中出现的两大类优化问题，这些问题会影响到编译程序的设计：（1）内存与寄存器分配问题；（2）循环分析问题。循环优化问题在很大程度上与机器的体系结构无关。存储分配和循环分析还涉及各种各样的子问题。

缩略词

CSE	Common Subexpression Elimination, 消除公共子表达式。
DAG	Directed Acyclic Graph, 有向无环图。
PLV	Persistent Loop Variable, 持久循环变量。

关键术语

code generation (代码生成) 产生可执行的代码, 其语义与输入源程序相同。

computer pipeline (计算机流水线) 它用于执行一条指令的时间被划分为离散的时间段, 并将这些时间段安排到硬件的顺序部件中。

induction variable (归纳变量) 任意普通变量或临时变量, 其值与循环的控制变量保持线性关系。

pipelined branch (流水线分支) 在分支指令生效之前, 计算机开始执行分支指令之后的下一指令。

registers (寄存器) 少量非常快速、且易于访问的内存。

address register (地址寄存器) 用于保存主存的地址。

data register (数据寄存器) 用于保存中间计算结果的值。

vector processor (向量处理器) 该处理器的设计是将单条或少量指令用于一个值的序列, 这些值要么规则地分布在内存中, 要么预先装入到向量寄存器中; 产生的结果存放在另一内存数组或向量寄存器中。

virtual memory (虚拟内存) 用于模拟和扩展主存的磁盘文件, 但是速度较慢。

练习

- 使用代码清单 9.3 所示的技术, 将下列待编译的每一条语句从 **IBSM** 零地址代码转换为寄存器机器代码。
 - $a := c * (a + b - c)$
 - $a := (a + b) * c - 3$
 - $a := c * (a - c) + d * (a + b)$
 - $a := (a + b) * (a - b)$
- 不要直接从零地址代码转换为一个虚构机器的代码, 而是(在纸上)模拟以下语句的代码生成程序并给出单地址代码, 其中消除了寄存器间接装入和存储指令(用直接装入和存储指令取代这些指令相应的地址装入指令)。
 - 代码清单 9.3
 - 练习 1 的 (a) 小题
 - 练习 1 的 (b) 小题
 - 练习 1 的 (d) 小题
- 使用图 9-2 所示的技术, 在编译时的栈中模拟练习 1 每一小题的代码执行。
- 使用图 9-3 所示的技术, 给出虚拟栈中的内存到内存代码, 模拟执行练习 1 中赋值语句的算术运算指令。你的答案应足够详尽, 并明确指出虚拟栈中每一变量的值。
- 证明在一棵结构化程序树中, 一次遍历的循环分析是安全且充分的。“一遍”循环分析是安全的, 如果认定所有循环变量的范围不会小于它们真正运行时的值范围; 它是充分的, 如果极少认定循环变量的范围远远大于它们真正运行时的范围。提示: 区别在循环入口处活跃的持久循环变量和入口处不活跃的持久循环变量。
- 给出一个分支寻址决策最优算法的伪码形式。
- 证明练习 6 的最优算法的复杂度为 n^2 , 其中 n 是分支的数目。
- 一个特殊的 **Pascal** 编译程序记住一个寄存器中装入了哪个变量或地址, 有时可避免重新装入这一个值。该编译程序为每一寄存器指派了一个值函数, 该函数估算寄存器中的内容被再次引用的概率 [Jacobi, 1982, 第 12 页]。试为该编译程序提出一种改进的优化算法。提示: 考虑如何才可避免寄存器的存储操作; 还要引入一种技术, 用于取代寄存器编号的随机赋值。
- 什么优化技术可用于处理抽象数据类型?
- 何种形式的程序统计分析可用于在编译时决定不压缩一个集合变量?

复习小测验

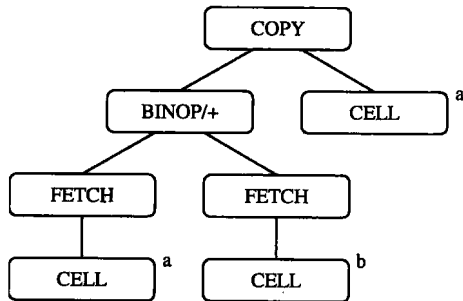
指出下列陈述是否正确。

1. 一个程序的大多数执行时间花费在 I/O 例程和外循环。
2. 循环展开仅限于未指定迭代次数的短循环。
3. 如下形式的 **WHILE** 循环可部分地展开：

```
WHILE BooleanCondition DO
    statement
END
```

4. 对于恰好横跨一个多维数组的一系列嵌套循环，展开其外循环等价于将数组声明重新解释为一个一维数组。
5. 寄存器分配本质上是一个图着色问题。
6. 图着色充分描述了将寄存器分配给表达式求值的问题。
7. 一个不含数据流分析的简单“一遍”编译程序，可通过在寄存器中模拟表达式栈，为一个多寄存器机器完成合理的代码生成。
8. 一个向量化编译程序寻找机会将内层 **FOR** 循环转换为向量代码，只要循环中所有对循环出口处仍活跃的变量的赋值是由下标指定的数组元素或是约简变量，并且所有数组下标是循环不变的或线性的。
9. 以下 7 个结点的树片段生成单条机器指令：

```
MOV b, a
```



编译程序实验项目

1. 为你的编译程序添加循环优化。
2. 选择一种你熟悉的向量计算机，修改你的编译程序，为该计算机生成向量化的代码。

进一步阅读

- Aho, A.V., Ganapathi, M., & Tjiang, S.W.K. "Code Generation Using Tree Matching and Dynamic Programming." *ACM Transactions on Programming Languages and Systems*, Vol.11, No.4 (October 1989), pp.491-516.
- 引入一种树操纵语言，用于构造高效的代码生成程序。
- Chaitin, G.J. "Register Allocation and Spilling Via Graph Coloring." *SIGPLAN 82 Symposium on Compiler Construction* (1982), pp.98-105.
- Chaitin, G.J. et al. "Register Allocation Via Coloring." *Computer Languages*, Vol.6, No.1 (1981), pp.47-57.
- Chow, F. & Hennessy, J. "Register Allocation by Priority-Based Coloring." *SIGPLAN 84 Symposium on Compiler Construction* (1984), pp.222-232.

Jacobi, C. *Code Generation and the Lilith Architecture*. Ph.D. dissertation, Swiss Federal Institute of Technology, Zurich, 1982.

介绍 Lilith 计算机是如何设计的, 从而可采用 Modula-2 语言编程。

Kennedy, K. "Index Register Allocation in Straight Line Code and Simple Loops." *Design and Optimization of Compilers* ed. R. Rustin. Englewood Cliffs, NJ: Prentice Hall, 1972, pp.51-64.

Knuth, D.E. "An Empirical Study of FORTRAN Programs." *Software – Practice and Experience*, Vol.1, No.2 (1971), pp.105-133.

发现大多数的执行时间花费在 I/O 例程和内循环。

Pittman, T. *Practical Code Optimization by Transformational Attribute Grammars Applied to Low-Level Intermediate Code Trees*. Ph.D. dissertation, University of California at Santa Cruz, 1985.

参阅附录, 其中给出了实现一个完整优化程序所需的 15~20 种文法中的 9 种代表性文法。

第 10 章 非过程式语言

本章旨在：

- 考虑应用式语言的编译相关问题
- 处理将函数递归转换为循环结构的问题
- 为 Itty Bitty 栈机器开发一个应用式程序设计语言（Tiny Scheme）的实现
- 探讨与基于 TAG 的“编译程序-编译程序”实现相关的课题
- 将变换属性文法看作一种用于定义编译程序的非过程式程序设计语言
- 介绍 TAG 的四个组成部分
- 将 TAG 看作一种不含变量或赋值语句的数据流语言
- 思考由一个 TAG 编译程序构造的有穷状态自动机

10.1 简介

本书的前 9 章仅讨论了诸如 Modula-2 等过程式程序设计语言的编译问题。这些技术还能以同样的方式应用到大多数常见的程序设计语言，故没有必要为教学目的再引入其他语言。本章将关注一些特殊的问题，这些问题关系到如何将其他类别的程序设计语言编译为高效的本地机器代码。编译技术有趣且可发挥作用的非过程式语言有三大类，即类似于 Lisp 或由其派生语言的函数式或应用式语言、所谓“第四代”数据库语言以及与 Prolog 语言有关的不确定推理语言。本章仅考虑其中一种语言的扩展例子，揭示编译程序设计原理是如何应用到这类语言的。

尽管普遍认为面向对象程序设计（OOP）是程序设计方法学的一个重大进展，但这类语言并未给编译程序设计人员带来特别的挑战。将信息封装在“对象”之中只是简单扩展了本书第 5 章已讨论过的作用域规则的编译单元，相当于在 Modula-2 语言中管理多个模块。对象类是包含了过程指针（类中的方法）作为字段的记录类型；在符号表中，这些记录定义所附带的信息包括了过程的引用，它们指向的过程将用于初始化该作用域级别的过程指针。单继承意味着记录类型在一个新的作用域中是可扩展的；当一个类定义为另一个类的子类时，在添加新的字段之前会将所有的字段定义导入新的记录定义中。重定义一个方法意味着当该类的一个对象被实例化时（即创建一个该记录类型的动态变量），一个局部过程可能替换父类中定义的一个过程。多态性是指一个过程（或运算符）能接受的参数可以是预定义集合中的任意类型。Modula-2 语言中运算符“+”是多态的，因为其操作数可能是 **INTEGER**、**REAL** 或集合，我们很容易在属性文法中实现这种多态性。多态性通常被认为是 OOP 的一个显著特点，但往往仅限于 OOP 中的一个方法能接受该方法所在类的继承链中的参数。因而 OOP 语言的实现与实现 Modula-2 语言的设计复杂度几乎相同，本书除此段简短评述之外，不再专门赘述这些问题。

最后，所有编译原理教材都会讨论编译程序的自动构造技术。本书所介绍的许多技术均使用了变换属性文法（TAG）作为实现的载体。尽管第 5 章介绍了一种确定的翻译方法可将一个 TAG 转换为过程式代码，但文法在本质上并不是过程式的。在本章、同时也是本书的结尾，还讨论了基于 TAG 的“编译程序-编译程序”的实现问题。

10.2 应用式语言的编译

1958年, John McCarthy 提出 Lisp 程序设计语言, 并作为麻省理工学院 (MIT) 人工智能研究项目的一部分 [McCarthy, 1981]。Lisp 是一种应用式语言, 它依靠对函数的应用, 而没有赋值语句。编译一种应用式语言本质上类似于编译一种过程式语言的表达式求值和递归函数, 但其中有一个重要的区别。不同于传统语言, Lisp 及其派生语言允许直接操纵程序代码, 这就产生了“延拓”(continuation)或部分求值函数, 它们可以像数据那样被操纵。在交互性较强的程序开发环境中, 通常推荐使用增量编译程序, 为用户修改可执行代码提供更短的周转时间; 如果一种语言允许或鼓励动态地修改代码, 则对编译程序也会有类似的需求, 因为新的(未编译的)代码可在运行时创建。这类系统中的编译程序必须作为运行环境的一部分。

本书第1章揭示了编译程序与解释程序的区别。编译程序将源代码翻译为另一种语言, 通常是目标计算机的机器语言, 但并不运行该程序; 解释程序可能根本不作翻译, 而只是直接地执行源程序中的语义动作。在一个交互式系统中, 语言的使用者看到的差别会少一些, 因为增量编译程序可具有与解释程序相同的响应能力, 同时取得与编译程序相同的某些性能优势。

编译时对函数的部分求值主要是将常量折叠变换应用到程序代码中。鉴于越来越强调函数的应用, 我们将研究如何从被调用函数的某一类调用中对常量参数进行折叠, 为每一个不同的常量值创建一个参数更少的函数。将一个或多个参数折叠为函数定义, 从而只需提供剩余的参数即可调用这些函数, 这一过程称为柯里化(currying); 这一术语以逻辑学家 Haskell Curry 的名字命名, 他改进了 Moses Schönfinkel 的最初想法 [Curry, 1968; Schönfinkel, 1924]。在编译程序中, 如果一个函数的大多数调用都使用了常量作为实参, 最有价值的做法是让其形参接受较少数目的实参值。在柯里化(折叠后)的每一个函数副本中, 新的常量参数往往还会产生另外的常量折叠机会。这一优化技术也可用于传统语言的编译程序, 但其优势就远不如用于函数式语言。

在 Lisp 语言的早期实现中, 函数定义中的非局部变量(也称自由变量)采用动态作用域。在纯应用式语言中不存在赋值语句, 因而除参数之外不存在其他局部变量; 一个对非参数的标识符的引用, 将指向调用链中该名字的最近的定义, 而不是参照源程序文本中的位置。一个自由变量引用的语义在很大程度上依赖于该函数的调用者。与此相反, Modula-2 和大多数块结构的程序设计语言采用静态作用域, 一个自由变量的语义仅依赖于源程序文本中该函数所处的位置。尽管动态作用域貌似更强大一些, 但在实践中却是得不偿失。如果采用静态作用域, 编译程序在编译时只需几条机器指令即可将所有自由变量链接到其引用(参阅第6章关于 Display 表的介绍); 而动态作用域则要求在运行时的执行环境中保存并查找符号表。随着编译型 Lisp 语言及其派生语言越来越流行, 动态作用域已逐渐失宠。

解释型语言比编译型语言更难实现的另一常见特点是动态数据类型。大多数计算机采用不同的硬件指令处理浮点数和整数, 每种数据类型的大小也不相同。最高效的机器代码是由编译程序根据待处理的数据类型选择了合适的硬件指令, 这就要求每一数据的类型在编译时是已知的。经典的程序设计语言(例如 FORTRAN 和 BASIC)通过语法形式(命名习惯)区别数据的类型; 诸如 Modula-2 等大多数现代语言则要求先将变量声明为某一特定的类型, 并在编译时将这些信息存储到符号表中, 供程序语义分析和代码生成时使用, 如本书第5、6章所述。由于在没有编译程序的情况下 Lisp 语言得到长期且广泛的应用, 这助长了 Lisp 语言的发展过程

中避开数据类型声明的需求，最终导致所有通情达理的编译程序不得不处理动态数据类型问题，编译得到的代码必须检查数据，以确定应用哪种合适的算术运算硬件指令。采用与消除范围检查代码相同的方法，利用充分的数据流分析也可消除一些类型分析代码；但值得怀疑的是，是否真的有这么多种编译程序真正地试图执行这类优化。

10.2.1 Lisp 语言的一些概念

为正确理解一种应用式语言的实现，我们需要熟悉使用该语言时的一些程序设计概念。对于曾使用过 Lisp 或其派生语言的读者而言，本小节的介绍很容易理解；但对于无此类经验的程序员而言，一些最基本的概念将有助于对问题的理解。

尽管 Lisp 语言的名字是从短语 **LISt Processing**（表处理）中抽取出来的，但 Lisp 之类的语言的基本数据结构是单元，即点对。这实际上是由两个指向同一类型的指针组成的记录类型，由此可构造出任意的二叉树结构。一个表由沿右子树向下递归连接的点对构成，如图 10-1 所示。

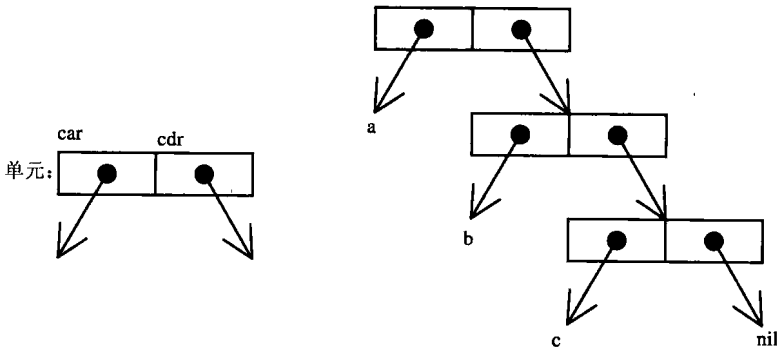


图 10-1 根据点对构造一个表 (a b c)

一个原子是单个值，通常是一个作为标识符的字符串，或是一个数字。一个点对中的每一分量要么指向另一点对，要么表示一个原子。表的构造就是将点对构造函数 **cons** 应用到两个参数上，这些参数本身又是一个点对或一个原子。另两个内置函数 **car** 和 **cdr**（分别读作 **car** 和 **could-er**）分别从一个点对中提取左子树和右子树。

表不仅可用于构造任意的数据结构，还可用于定义程序代码。表 (a b c) 解释为一个程序片段时，是一个函数的调用或应用，表示将一个名为 **a** 的函数应用到两个参数 **b** 和 **c**。由于 **b** 和 **c** 本身可能也是表，因此程序可以是相当复杂的。更特别的是，被应用的函数本身可能是一个表，因而（至少在一个正确的 Lisp 程序中）参数会被传递给一个作为函数调用返回值的函数。这在 Modula-2 语言中也是可能的（当一个过程返回的结果是一个过程类型时），但 Modula-2 语言不允许直接调用返回的过程，而要求将它先赋值给某个变量。纯 Lisp 语言没有赋值语句，并且对一个程序表的求值结果是某一表达式的值。表达式（函数的应用）被求值，其结果作为参数再传递给其他函数，如此递归地执行，直至整个程序表求值完毕。最终的结果通常打印（显示）在用户终端上。

除 **car**、**cdr** 和 **cons** 之外，Lisp 语言还有一个内置函数用于创建一个函数，称之为 **lambda**；该名字依据作为 Lisp 语言数学基础的 λ 演算命名。在这里讨论的 Lisp 语言的派生语言 Scheme 中，**lambda** 接受两个参数，第一个参数是用于命名新函数的参数的标识符表，第二个参数是一个表的表达式（通常在其中用到那些参数）。故 **(lambda (x y) (+ x y))** 是一个求值结果为函数的表达式，以两个参数调用该函数时，它将返回这两个参数之和。函数

`(lambda(x) x)` 是一个恒等函数，即传递给该函数任何参数都将返回参数本身。将参数传递给一个函数，相当于为函数体的值指定了一个名字。据此，可将一个恒等函数的定义应用到某一函数定义，从而形成递归函数；该递归函数成为恒等函数中一个参数，但在此过程中它得到了一个名字，从而可以调用其自身。返回函数的函数是一个较难掌握的概念，仅由递归复合而成。然而一旦完全理解了这一概念，就会发现这个概念的优雅和强大。

其他的内置函数执行标准的算术运算（例如上述例子中的“+”），允许一个表不解释为函数调用就写入程序中（称为“引文”）以及允许子表达式的条件求值（`if`）。以下 Lisp 程序是一个阶乘函数，如果其参数为 0 则返回 1，否则用参数乘以一次递归调用的结果：

```
((lambda(fact) fact) (lambda(n) (if (= n 0) 1 (* (fact(- n 1))))))
```

大多数 Lisp 语言的派生语言还预定义了若干函数，这些函数实际上是基本λ演算形式的语法扩展。其中一些函数表面上看起来好像赋值语句和语句序列，但可证明其中的大多数等价于λ表达式。例如，以下表片段中的函数 **define** 看起来类似一条赋值语句，因为它将值 3 赋给后续表项目中的名字 **x**：

```
... (define(x 3)) (+ x 1) ...
```

而同一表片段可等价地写作一条如下的λ表达式：

```
... ((lambda(x) (+ x 1)) 3) ...
```

10.2.2 尾递归

应用式语言中缺少类似 **WHILE** 循环这种执行次序控制结构，这意味着所有迭代必须通过函数的递归来完成。编译程序当然希望在可能的情况下将一个递归转回一个循环结构，在传统语言中也可能有这种优化，但远没有这么重要。在尾递归的情况下，这种转换相当直接：所谓尾递归，就是指计算的最后一步是一个递归调用，其结果将不加修改地返回给先前的调用者。为演示这一概念，考虑以下阶乘函数（虽然采用 Modula-2 语言书写，但这段代码完全是应用式的）：

```
PROCEDURE fact(n: INTEGER): INTEGER;
BEGIN
  IF n = 0 THEN
    RETURN 1
  ELSE
    RETURN n * fact(n - 1)
  END
END fact;
```

尽管上述函数的结尾是对函数自身的递归调用，但这并不是计算的最后一步，因为结果向上传递之前还必须乘以 **n**。该函数不是尾递归的，但可改写为如下的尾递归形式：

```
PROCEDURE fact(n: INTEGER): INTEGER;
  PROCEDURE fact2(n, m: INTEGER): INTEGER;
  BEGIN
    IF n = 0 THEN
      RETURN m
    ELSE
      RETURN fact2(n - 1, n * m)
    END
  END
```



```
END fact2;  
BEGIN (* fact *)  
    RETURN fact2(n, 1)  
END fact;
```

现在内层函数 **fact2** 不加修改地返回其自身的递归调用结果，所有乘法均已在函数调用的参数中完成，而不是在函数的结果中。只要将包含递归调用的整行语句替换为直接跳转到函数开头的跳转语句，就很容易将这类尾递归转换为传统的循环结构；新的值将直接取代当前的参数，而不是将调用的参数存储到一个新的激活帧中。编译程序必须极为小心地避免当一个参数仍活跃时（即在所有新的值被计算出来之前）重写该参数，但可将新的值赋给临时变量，然后应用第 8 章介绍的传统数据流分析和复写传播技术消除不必要的代码迁移。从源代码变换的角度看，转换结果如下：

```
PROCEDURE fact2(n, m: INTEGER): INTEGER;  
BEGIN  
    LOOP  
        IF n = 0 THEN  
            RETURN m  
        ELSE  
            m := n * m;  
            n := n - 1  
        END  
    END (* LOOP *)  
END fact2;
```

当优化程序将函数 **fact2** 回代到其函数调用中之后，结果等价于如下代码；当然这再也不是应用式代码了，但这正是我们的目标：

```
PROCEDURE fact(n: INTEGER): INTEGER;  
VAR  
    m: INTEGER;  
BEGIN (* fact *)  
    m := 1;  
    WHILE n > 0 DO  
        m := n * m;  
        n := n - 1  
    END;  
    RETURN m  
END fact;
```

我们并不是真的指望编译程序将一个非尾递归函数转换为尾递归的；但一旦发现存在尾递归，很容易将它优化为一个循环。

10.2.3 实现一个应用式语言的编译程序

一个很少被颂扬的优点是自编译的编译程序，即一个编译自身的编译程序。乍看起来，自编译的意义在于能够帮助解决自举问题；实际上这种类比是恰当的，术语“自举”通常用于表示让编译程序或操作系统到达某个点，从这一点起它们自己就可以运作。对于编译程序而言，第一步就是在另一个环境中编译一个可能简化了的编译程序版本；这可能是使用另一种计算机上已有的同一种语言，又或者是将编译程序手工编译为另一种可用的语言。然后第一个编译程序就可用于将自身编译为目标机器代码，接着最后在目标机器上用编译好的编译程序编译其自身。

如果自举时采用了简化版本,则此时该语言可扩展为覆盖其全部能力,并重编译增强版本。如果这一最终阶段还包括了早期版本省略的代码优化工作,那么连续再次对编译程序的重编译(每次使用新编译的版本编译下一版本)还可提升编译程序自身的性能。然而,须注意细微代码错误的熵效应,它们传播起来就好像生物学中对应的“突变”一样,只需几代就可令重编译的编译程序沦为不育的或者无用的。

采用语言本身实现编译程序的主要优点是,鉴于编译程序的大小和复杂度,它能够对自己的大多数代码进行严格测试,从而在开发过程中比那些为测试目的而创建的玩具似的程序有可能捕获更多的错误。由于开发人员被迫使用自己的创造成果,这也激励了他们将作品完成得更便于使用、更高效,相当于在一个更高的抽象层次获得同一优点。因而,大多数 Lisp 编译程序采用 Lisp 语言编写就不足为奇了,事实上也理应如此。

本章稍后将重点讨论一个 TAG 编译程序的开发,它本身采用 TAG 书写。然而本书的重点一直是将属性文法用于编译程序的实现,因此在研究一个小型应用式语言的编译程序时,我们搁置自编译的编译程序目标,而选择完全以定义了语言的语法和语义的文法术语来讨论这一问题。

不管采用什么语言,编译程序设计的需求之一是精确地定义源语言和目标语言。为与本书之前的内容保持连贯性,我们仍将 Itty Bitty 栈机器作为目标语言;第9章已介绍了针对商用计算机设计编译程序时可能不同于面向 IBSM 的相关课题。IBSM 定义为一种传统的(整数)算术运算计算机,而 Lisp 语言的基本数据结构是表(或更精确地说,是点对)。因而,有必要定义点对如何翻译为本地的 IBSM 数据结构,为此我们选用与以下 Modula-2 语言的记录类型等价的机器码表示:

```

TYPE
  Cell = POINTER TO Datum;
  Lambda = PROCEDURE(paramlist: Cell): Cell;
  TagType = (code, actf, pair, atom, intn);
  Datum = RECORD
    CASE tag: TagType OF
      code:
        fun: Lambda;
        nArgs: CARDINAL;
        itsEnv: Cell |
      pair:
        ar, dr: Cell |
      iden:
        offs, lex: INTEGER |
      atom, intn:
        val: INTEGER
    END
  END;

```

这里令 **atom** 的变体包含一个索引,指向某一未详细说明的全局字符串表,类似于本书第3章讨论过的做法;还有更好的实现方法,但这里没必要让其复杂性分散我们的注意力。虽然我们只定义了一个数值数据类型(整数),但我们也可支持诸如有理数或大整数等其他类型,通过点对或整数表可构造出这些类型。只需简单的改动就可将这些数据类型添加为内置类型,即在 **tag** 类型中包含它们的枚举,并声明一种合适的机器级数据结构以支持它们。此处的点对结构建模了点对的早期形式(名字地址寄存器 **ar** 和递减寄存器 **dr** 系指早期实现中一个机器

指令字的组成部分)。

上述数据结构还为编译后的代码引用定义了一个变体。一个代码引用的组成包括：一个指向内存中实际机器代码的指针（类型为 **Lambda**）、一个指向其符号表（在该作用域中需要编译更多的代码时将用到）的引用以及被调用时期望在其激活记录中找到的参数的个数。根据同一记录中的符号表引用，可在运行时的栈中构造激活记录本身，表中的每一单元对应一个指定的参数；因而激活记录的大小取决于函数的定义。

我们选择 Scheme 语言的一个很小子集作为源语言；Scheme 语言是 Lisp 的派生语言，具有静态作用域和相当规则的结构与原语 [Sussman&Steele, 1975; Smith 1988]。假设编译程序包含在一个交互式的环境中，因而只要有未编译代码的函数被应用，编译程序就将被调用。未编译的代码总是以表的形式出现，即一个右递归的点对链表。我们开发了两个文法，一个分析源程序文本并转换为表，另一个将指定的表编译为 **IBSM** 代码。尽管对内置函数库的技术细节的深入研究超出了本书的讨论范围，但有一点很重要，就是需区分那些由编译程序识别并编译为内联代码的函数和那些直接作为全局定义的代码而存在的函数。为此，我们淡化了执行环境与编译程序之间的边界，因为正在运行的程序可创建新的表，并将它们提交编译。在实际应用中，这是编译程序被激活的惟一途径，因为在用户交互最顶层的情况也是如此。

一个 Lisp 解释程序的核心是表达式求值程序。如果提交的是一个原子，求值程序将在符号表中查找该原子，并返回其值；如果是一个点对，求值程序将对 **car**（即上述记录类型中的 **ar** 字段）求值，期望找到一个函数引用，并将 **cdr**（记录类型中的 **dr** 字段）作为参数表传递给该函数。在本书的实现中，求值程序调用编译程序，并将整个表达式传递给编译程序，然后执行作为结果的编译后代码。编译程序根据代码清单 10.2 的文法分析这一表达式树；代码清单 10.1 则分析源程序文本，并转换为表单元的树表示。如果读者已熟悉 Scheme 语言（或 Lisp 的某一其他派生语言），这些文法将更容易理解。还须注意的是，编译程序文法发送给代码生成程序的代码 **EmitIBSM** 通常将两条或三条指令压缩为一个字，以提高紧凑性；附录 C 解释了这在 Itty Bitty 栈机器中是如何工作的。

代码清单 10.1 将 Tiny Scheme 语言从源程序转换为树表示的文法

```

SchemeTree ↑theTree:tree

→ "(" aList ↑theTree ")"

→ ATM ↑strno                                { ATM 返回扫描程序的一个单词 }
  [theTree = <atom>%strno]

→ NUM ↑valu                                { NUM 是扫描得到的一个数字 }
  [theTree = <intn>%valu]

→ "" SchemeTree ↑aTree
  [theTree = <pair <atom>%-2 aTree>]
;

aList ↑theTree:tree

→ SchemeTree ↑aTree

```

```

    ("." SchemeTree ↑bTree
      [theTree = <pair aTree bTree>]
    | aList ↑bTree
      [theTree = <pair aTree bTree>]
    )

→ [theTree = <>]                                { 返回 nil }
;

```

代码清单 10.2 Tiny Scheme 语言的代码生成文法片段

CompileExpn ↓cantail:bool ↓env:tree ↓inCode:tree ↑outCode:tree

```

→ <pair <atom>%n rite>
  [n < 0]                                { 索引为负数表示一个内置函数 }
    ([n + 1 = 0]
      rite: DoLambda ↓env ↓inCode ↑outCode
    | [n + 2 = 0]
      rite: DoSequence ↓cantail ↓env ↓1 ↓0 ↓inCode ↑outCode
    | [n + 3 = 0]
      rite: DoCond ↓cantail ↓env ↓inCode ↑outCode
    | [otherwise]
      rite: DoArith ↓env ↓-n ↓inCode ↑outCode
    )

→ <pair <atom>%n rite>
  [n > 0]                                { 索引为正数表示一个原子 }
    [Lookup ↓env ↓n ↑theTree]            { 找出它的当前定义 }
    ([theTree: <pair ...>] | [theTree: <iden>] | [theTree: <code ...>])
      [aTree = <pair theTree rite>]       { 用它替代该原子 }
      aTree: CompileExpn ↓cantail ↓env ↓inCode ↑outCode
                                              { 再次尝试 }
    | [otherwise]
      rite: CompileList ↓env ↓inCode ↑aCode { 自由变量作为栈中的参数 }
      CallLibe ↓env ↓3 ↓n ↓aCode ↑fCode   { 添加库查找引用 }
      CallLibe ↓env ↓2 ↓cantail ↓fCode ↑outCode
                                              { 调用它的库引用 }
    )

→ <pair <code theCode> rite>             { 先前已编译的代码 }
  [GetInfo ↓theCode ↑itsenv ↑nargs ↑addr]
  rite: DoSequence ↓false ↓env ↓1 ↓nargs ↓inCode ↑aCode
                                              { 取参数 }
  [EmitIBSM ↓894 ↓aCode ↑bCode]          { 将父帧指针压入栈中 }
  ([itsenv > 0; EmitIBSM ↓889 ↓bCode ↑bCode@; itsenv@ = itsenv - 1])*
  ([cantail = true]
    TailCall ↓theCode ↓bCode ↑outCode
  )
  | [otherwise; EmitIBSM ↓124 ↓bCode ↑cCode]
                                              { 将尾递归调用转换为跳转语句 }
  [EmitIBSM ↓addr ↓cCode ↑outCode]       { 生成子例程调用 }
  )

→ <pair left:<pair ...> rite>             { 未编译的函数调用 }

```

```

rite: CompileList ↓env ↓inCode ↑aCode      { 将代码生成到栈参数表中 }
left: CompileExpn ↓false ↓env ↓aCode ↑fCode { 取函数代码 }
callLibe ↓env ↓2 ↓cantail ↓fCode ↑outCode  { 添加库引用以调用它 }

→ <iden <intn>%lex>%offs                    { 局部参数标识符 }
  [CurrentLex ↓env ↑cur]
  ([lex = cur]                                { 如果词法层次是当前层次, }
    [EmitIBSM ↓892 ↓inCode ↑aCode]           { 生成代码将它的值压入栈中 }
    [EmitIBSM ↓offs ↓aCode ↑outCode]
  |[otherwise; n = cur - lex]                 { 否则, 是一个非局部的引用 }
    [EmitIBSM ↓28633 ↓inCode ↑aCode]         { 生成跟在静态链后面的代码 }
    ([n > 0; EmitIBSM ↓827 ↓bCode ↑bCode@; n@ = n - 1])*
    [EmitIBSM ↓28059 ↓bCode ↑cCode]         { 生成代码将它的值压入栈中 }
    [EmitIBSM ↓offs ↓cCode ↑outCode]
  )

→ <atom>%n                                  { 假设是索引是一个正数 }
  [Lookup ↓env ↓n ↑aTree]                   { 找出它的当前定义 }
  (([aTree: <pair ..>] | [aTree: <atom>] | aTree: <intn>) | [aTree: <code>])
  aTree: CompileExpn ↓cantail ↓env ↓inCode ↑outCode { 再次尝试 }
  |[otherwise; Error]                        { 未定义的符号引用 }
  )

→ <intn>%n                                  { 文字量数字 }
  [EmitIBSM ↓28 ↓inCode ↑aCode]             { 生成代码将它的值压入栈中 }
  [EmitIBSM ↓n ↓aCode ↑outCode]

→ <code theCode>                            { 代码引用 }
  [GetInfo ↓theCode ↑itsenv ↑nargs ↑addr]
  [EmitIBSM ↓28 ↓inCode ↑aCode]             { 生成代码将函数引用压入栈中 }
  [EmitIBSM ↓addr ↓aCode ↑outCode]
;

```

DoArith ↓env:tree ↓op:int ↓inCode:tree ↑outCode:tree

```

→ <pair left <pair rite <>>>                { 需要两个参数组成的表 }
  left: CompileExpn ↓false ↓env ↓inCode ↑aCode { 对参数求值 }
  rite: CompileExpn ↓false ↓env ↓aCode ↑eCode
  rite: DoArithOp ↓op ↓eCode ↑outCode        { 生成运算符的代码 }

→ atree                                      { 未编译的参数表, 故 ... }
  atree: CompileList ↓env ↓inCode ↑aCode     { 在栈参数表中生成代码 }
  [EmitIBSM ↓13288 ↓aCode ↑bCode]           { 生成提取 car 并压入栈中的代码 }
  [EmitIBSM ↓10105 ↓bCode ↑cCode]
  [EmitIBSM ↓12381 ↓cCode ↑dCode]           { 生成提取 cdr 的代码 }
  [EmitIBSM ↓889 ↓dCode ↑eCode]
  rite: DoArithOp ↓op ↓eCode ↑outCode       { 生成运算符的代码 }
;

```

CompileList ↓env:tree ↓inCode:tree ↑outCode:tree

```

→ <pair left rite>
  left: CompileExpn ↓false ↓env ↓inCode ↑aCode      { 编译表的 car }
  rite: CompileList ↓env ↓aCode ↑cCode              { 执行 cdr }
  CallLibe ↓env ↓1 ↓0 ↓cCode ↑outCode              { 将库引用添加到 cons }

→ <>
  [EmitIBSM ↓30 ↓inCode ↑outCode]                  { 在运行时将 nil 压入栈中 }
;

```

DoArithOp ↓op:int ↓inCode:tree ↑outCode:tree

```

→ atree                                     { 操作数已编译好 }
  ([op = 4; EmitIBSM ↓12 ↓inCode ↑outCode] { 加法 }
  |[op = 5; EmitIBSM ↓404 ↓inCode ↑outCode] { 减法 }
  |[op = 6; EmitIBSM ↓11 ↓inCode ↑outCode] { 乘法 }
  |[op = 8; EmitIBSM ↓16 ↓inCode ↑outCode] { 判断相等 }
  |[op = 9; EmitIBSM ↓17 ↓inCode ↑outCode] { 判断小于 }
  |[otherwise; Error]                       { 未知运算符 }
  )
;

```

DoSequence ↓tail:bool ↓env:tree ↓cnt:int ↓limit:int ↓inCode:tree ↑outCode:tree

```

→ atree
  [count = limit]                             { 如果相等则仅有一个表的值, }
  atree: CompileList ↓env ↓inCode ↑outCode { 因而生代码放入栈中 }

→ <pair left rite>
  left: CompileExpn ↓(tail&rite=<>) ↓env ↓inCode ↑aCode { 编译 car }
  rite: DoSequence ↓env ↓cnt+1 ↓limit ↓aCode ↑outCode { 执行 cdr }

→ <>
  [outCode = inCode]
;

```

AddSymbols ↓inenv:tree ↑outenv:tree ↑nsyms:int

```

→ <pair <atom>%n rite>                      { 所需的符号 }
  rite: AddSymbols ↓inenv ↑anenv ↑syms        { 计算它们的个数以取得偏移量 }
  [CurrentLex ↓anenv ↑lex; offs = -2 - syms; nsyms = syms + 1]
  [into ↓anenv ↓n ↓<iden <intn>%lex>%offs ↑outenv]
                                          { 插入该名字 }

→ <>
  [outenv = inenv; nsyms = 0]
;

```

DoLambda ↓env:tree ↓inCode:tree ↑outCode:tree

```
→ <pair left rite>                                     { 需要两个表 }
  [OpenFrame ↓env ↑newenv]                             { 为参数开设新的帧 }
  left: AddSymbols ↓newenv ↑funenv ↑nsyms               { 插入它们 }
  [EmitIBSM ↓158 ↓<> ↑bCode]                             { 生成过程的入口代码 }
  rite: DoSequence ↓true ↓funenv ↓1 ↓0 ↓bCode ↑eCode    { 生成过程体 }
                                     { 添加出口代码 }
  [EmitIBSM ↓188 ↓eCode ↑xCode]
  [EmitIBSM ↓nsyms ↓xCode ↑fCode]
  [MakeCode ↓funenv ↓nsyms ↓fCode ↑theCode]
  [GetInfo ↓theCode ↑itsenv ↑nargs ↑addr]
  [EmitIBSM ↓28 ↓inCode ↑cCode]                         { 生成LDC指令将引用压入栈中 }
  [EmitIBSM ↓addr ↓cCode ↑outCode]
;

DoCond ↓cantail:bool ↓env:tree ↓inCode:tree ↑outCode:tree

→ <pair expn <pair left <pair rite <>>>>               { 需要3个参数组成的表 }
  expn: CompileExpn ↓false ↓env ↓inCode ↑aCode          { 对布尔表达式求值的代码 }
                                     { 添加条件分支语句 }
  [EmitIBSM ↓60 ↓aCode ↑bCode]
  [EmitIBSM ↓0 ↓bCode ↑tCode]
  left: CompileExpn ↓cantail ↓env ↓tCode ↑xCode
                                     { 生成true分支的代码 }
  [EmitIBSM ↓1950 ↓xCode ↑cCode]                         { 无条件分支 }
  [EmitIBSM ↓0 ↓cCode ↑eCode]
  [BackPatch ↓bCode ↓eCode ↑fCode]                     { 条件分支到达此处 }
  rite: CompileExpn ↓cantail ↓env ↓fCode ↑gCode
                                     { 生成false分支的代码 }
  [BackPatch ↓cCode ↓gCode ↑outCode]                     { 无条件分支到达此处 }
;

;
```

编译程序已知的内置函数在符号表中标识为负数索引的原子。其中一个函数应用到一个合适的参数表，将产生本地的（内联的）IBSM 代码。文法中仅展示了若干此类函数，即整数算术运算符、条件与函数形式以及顺序结构（Scheme 语言中的 **begin**）。也可（并且应该）采用这一方法添加点对分解运算符 **car** 和 **cdr**；点对合成运算符 **cons** 需要内存分配，因而正确的做法仍是库函数调用，而不是纯内联代码。该编译程序已知三个库函数，由指向非终结符 **CallLibe** 的索引标识，表 10-1 也列出了这些索引。一个完整的实现可能还需要几个其他的内置函数和库函数，编译程序也必须知道这些函数。最后，有一些内存管理问题，但在这里与我们无关；完整的实现必须为编译好的代码分配内存结构，并实现一个合适的垃圾收集程序用于恢复不可访问的代码片段以及已经无用的表单元，它们的调用通常是在一个支持 **cons** 的库例程中。

表 10-1 Tiny Scheme 语言的关键内置例程与库例程

内置函数的索引	
-1	Lambda，用于生成函数代码
-2	顺序，也用于函数调用的栈参数
-3	条件，标准的 IF-THEN-ELSE 表达式形式

(续)

内置函数的索引	
-4	加法
-5	减法
-6	乘法
-8	比较是否相等
-9	比较是否小于
库例程的索引	
1	根据左部和右部 (car 和 cdr) 构造一个单元 (cons)
2	构造一个激活帧, 并调用任意一个函数。用于编译时无法确定函数的功能接口 (参数个数) 的情况, 例如在运行时才编译的函数, 或由一个自由变量查找得到的函数
3	运行时在符号表中查找一个自由变量

该文法省略了两个重要的非终结符。其中一个 **CallLibe**, 它只负责构造合适的代码以调用指定的库例程, 其细节可能依赖于将编译好的程序链接到库例程时所选用的机制。一种简单的实现是将例程库的入口点存放到内存低地址的向量中, 然后对库例程的调用可根据索引从该向量中取出一个项目。另一个未定义的非终结符是 **TailCall**, 它负责将一个尾递归的函数调用转换为跳转结构。在上述文法中, 这一优化的优势并不明显, 部分原因是由于 **IBSM** 的过程调用相对而言开销并不算大, 并且跳转指令的开销与之相比也相差无几; 另一部分原因是大多数新的激活帧总得先构造出来, 然后才可在旧激活帧上安全地复制参数值。如果当前被调用的过程的参数个数不同于调用者的参数个数, 栈中返回地址的位置也必须移动。完成这些任务的非终结符 (以及支持该非终结符的库例程的定义) 留作练习。

上述文法描绘的编译程序并未涉及延拓, 因为并没有为延拓生成技巧性较高的代码。**Scheme** 语言采用库函数 **call/cc** 获取对延拓的访问, 并且所有实现均隐藏在该库例程中。对一个延拓的求值效果是以一个新的结果跳转到 **call/cc** 的出口; 因而, 库例程必须在一个数据结构中保存运行时栈的内容, 以及每次对延拓求值时根据该数据结构重建栈的代码。这种情况可能在原有的栈消失很久后反反复复地出现, 因而必须保存整个栈的内容。当不再有任何引用指向该延拓时, 垃圾收集程序将理所当然地丢弃该数据结构并回收其内存空间。注意, 保存和重建运行时栈的效率在很大程度上取决于先前将尾递归转换为迭代的优化工作, 因为每一迭代程序均消除了栈帧, 因而这些栈帧无需保存在延拓之中。

上述文法也不支持常量折叠。我们认为这只是一个很简单的改进, 也将它留给读者作为练习。**Scheme** 语言的应用式本性容易令人急于为函数的定义和调用生成代码。一种好的优化工作要求尽可能长时间地以表形式保留过程的定义, 从而常量折叠技术有机会将调用中的常量参数折叠为简化的代码。在一个实用的 **Lisp** 编译程序中, 开发一种合适的启发式方法判断何时执行此项工作是相当困难的。对于传统语言而言, 过程的定义通常太大, 故不宜回代到过程的调用程序中, 并且算术运算和逻辑运算根本就不是过程调用, 因而这一决策通常不是一个问题。

关于程序的部分求值已开展了大量的研究, 作为一种有效手段将一个通用算法转换为一个处理特定类问题的程序。许多这类研究取得了性能上的显著提升, 但它们仅考虑了采用 **Lisp** 派生语言书写的程序。在这一背景下, 部分求值相当于传统程序设计语言中的全局常量折叠和过程回代。由于 **Lisp** 语言没有专门的语言结构支持与特定应用有关的算法, 所以这些结构的构建必须基于底层嵌套和递归函数的应用。部分求值将抽象的 **Lisp** 程序简化成 **Modula-2** 程序员

一般不必苦思冥想即可编写的程度，只要提供了第 8、9 章讨论过的编译程序优化技术。仅当与性能相对较差的解释型或半解释型本地 Lisp 程序相比时，部分求值的 Lisp 程序才有令人瞩目的优势。

10.3 变换属性文法的编译程序

作为一个编译程序的规格说明，变换属性文法（TAG）是一种非过程式程序设计语言，意即它可作为一种完整地定义一个程序（即一个编译程序）的形式化源语言。因而，我们有理由构思一个编译程序，它可根据 TAG 生成一个可使用的编译程序。将一个文法编译为代码的程序称为“编译程序—编译程序”；YACC（即 Yet Another Compiler Compiler 的缩写）是一个著名的“编译程序—编译程序”例子，它根据一个上下文无关文法的规格说明自动地开发一个分析程序。在源文法中，YACC 通过嵌入的 C 语言代码片段指定语义，这些代码仅能有限地访问单个受限的属性。扫描程序也必须在一个单独的程序（Lex）中开发。关于如何基于不同的集成度开发一个高效、高产的“编译程序—编译程序”已开展了大量研究工作 [例如：Leverett 等，1979；Ganzinger 等，1977]，但由于本书的重点是 TAG，本章只讨论如何运用先前章节介绍的核心概念实现一个 TAG 编译程序。

该 TAG 编译程序设计为一个自编译的 TAG，将 TAG 翻译为标准 Modula-2 语言。预定义函数（通常可用于一大类编译程序中）设计为 Modula-2 语言的过程库；在已完成的编译程序最终被编译成目标代码时，这些过程被链接到编译程序中。编译程序的源文法以一种独立的机器可读形式提供，包括各种常见的工业用和科研用计算机上的目标代码。这里描述的编译程序的许多特性均基于成熟的技术，无需特别的创新。我们的目标是讨论一个真正可使用的编译程序实现，从而鼓励更多的研究与开发。尽管不断有研究仍在进一步改进 TAG 语言及其编译程序，附录 B 包含了 TAG 语言的完整语法，以及本章描述的 TAG 编译程序版本的部分语义；机器可读的文件通常在不断更新。

第 3 章介绍的算法可根据正则表达式机械地构造一个 FSA 扫描程序；第 4 章展示了如何根据 LL(1)文法机械地构造一个递归下降分析程序；第 5 章扩展了该分析程序，使之支持属性求值；第 6 章和第 8 章简要讨论了 TAG 中的代码生成和代码优化问题。现在我们将这些技术组合为一个将 TAG 翻译为一个可使用的编译程序的工具。正如第 1 章的概述（参阅图 1-1），一个编译程序需考虑六个组成部分。其中，扫描程序、分析程序和树变换程序（优化程序）这三个部分在待编译的源文法中分别以不同的小节表示；而约束程序和代码生成程序的主要组成是写在分析程序和变换程序部分的属性求值函数，而希望执行的窥孔优化工作以及支持字符串表和符号表的部分则隐藏在其中的预定义属性求值函数中。

10.3.1 TAG 编译程序的组成部分

一个 TAG 有四个语法组成部分，每一部分定义了编译程序的一个单独成分。如果某一部分未用上，则可省略不写。第一部分声明了在整个 TAG 中可见的全局属性、可用于构造一棵树的结点名字以及在 TAG 其他部分可调用的预定义函数库的功能接口。第二部分以保留字 **scanner** 开头，为扫描程序的所有单词定义了正则表达式；这些单词往往无法定义为单个用引号括住的字符串文字量，或者涉及特殊的扫描程序语义。空白和注释也以类似方式定义，采用保留字 **ignore** 作为扫描程序的一个非终结符。第三部分以保留字 **parser** 开头，在一个扩

展了属性和正则表达式运算符的上下文无关文法中，定义了编译程序的语法和静态语义约束。最后一部分以保留字 **transformer** 开头，在一个上下文无关的树变换属性文法中，定义了待编译的编译程序的所有代码优化和优化生成需求。

分析程序的语法规则接受扫描程序的单词作为其输入语言；变换程序的规则在树模板上操作，但允许分析程序的一个非终结符通过调用变换程序的规则执行局部的树变换。类似地，变换程序中的一个非终结符可调用分析程序中的一个空非终结符（即一个不读入任何扫描程序单词的非终结符），以执行不应用到任何树上的属性求值。

TAG 中任一部分的继承属性和综合属性均须显式地定义其类型，可使用以下五种类型名保留字之一：**int** 表示整数、**bool** 表示布尔真值、**set** 表示变长的位向量、**table** 表示定长的向量（譬如散列表）、**tree** 表示带装饰的树结构。至于单个非终结符规则中的局部属性，则由 TAG 编译程序根据其用法自动确定其类型。

文法在本质上是非过程式的，但正如本书第 4、5 章所示，一个属性文法可直接转换为 Modula-2 这类过程式代码；除扫描程序的单词所要求的从左到右求值次序之外，这些过程式代码也具有强制规定了文法求值次序的效果。TAG 的变换程序部分不会引用输入单词，故原则上变换程序规则中对于一个非终结符的引用可按与属性流相容的任意次序求值。实际上，在 TAG 编译程序的一个早期版本中，就重新组织了属性求值函数的次序以消除向前引用依赖关系；但用户发现这些文法（以及编译得到的代码）非常难以理解和调试，因而后来摒弃了这一特征。当前版本的 TAG 编译程序确实区分了属性求值函数（将先前未受限的属性限定为单个值）和属性断言（对已受限的属性应用更多约束）。如果选择或迭代这类正则表达式运算符的用法导致这一决策出现二义性，或应用到一个属性的首个断言未能将它限定为单个值，则会报告一个错误。类似地，一个传给非终结符的继承属性或一个从非终结符得到的综合属性在它们被传递的那一刻起，就必须完全受限为单个值。

10.3.2 文法中的迭代运算符

涉及迭代运算符的正则表达式扩展给属性求值提出一个特殊问题。TAG 中没有变量或赋值语句之类的东西，从这一点来说 TAG 是一种数据流语言。一个属性求值函数可将一个属性的值限定为单个值，但此后任何求值函数（原则上）都不可修改该属性的这个值；其他的求值最多只能为这个值添加更多的约束作为断言，譬如将这个值与其他属性表达式进行比较，或从另一非终结符的引用综合出一个相同的值。只要该断言的检测失败，就会导致整条规则失败，除非有另一选项没有同时失败。在 Scheme 或 Lisp 这类语言中，由于缺少对变量的赋值，所以不会产生特别的问题，因为不存在这种迭代运算符而只有递归结构。类似地，如第 2、5 章所示，无须借助于赋值语句，文法的递归即可实现迭代的概念以及属性值的序列化。但一个文法如何能够处理正则表达式的星号运算符（例如用一个属性对迭代次数进行计数）？我们打算从第 3 章提出的迭代运算的递归等价形式中寻找灵感。

考虑一个简单的文法 G ，同时将它表示为递归形式和正则表达式的迭代运算符形式：

$$G \rightarrow b$$

$$G \rightarrow a^* b$$

$$G \rightarrow a G$$

假设需要为文法 G 综合出一个属性 n ，用于计算一个串中单词 a 的个数。在递归形式中，可直接得到如下的属性求值：

$$\begin{array}{ll} G \uparrow n \rightarrow b & [n = 0] \\ \rightarrow a \ G \uparrow m & [n = m + 1] \end{array}$$

文法 G 的重写过程中，每一句型均创建了属性 n 的一个新实例，这些实例均有自己惟一的（固定的）值：0、1、2、…。非终结符的调用之间并没有赋值的意思，因为 n 的单个值在这一递归层次的整条规则中都是有效的。值的稳定性是一个关键的概念，如图 10-2 中的层次所示。属性 n 在它的层次中有固定的值，但综合出来并向上传给下一层次的值可能不同。

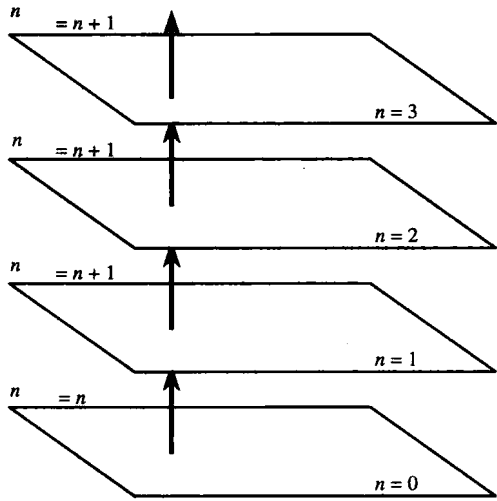


图 10-2 横跨了递归或迭代的属性求值

此时，令图 10-2 中的每一层次表示一次迭代，而不是递归。在该迭代中，局部属性 n 仍有固定的值，并且向上传递给下一迭代的值也可能不同。我们可在文法的语法上表示将局部属性的一个新固定值传递给下一迭代，即在属性名之后附加一个符号“@”。该符号反映了属性值的传递穿越了迭代的边界，并且不应与变量的赋值相混淆；变量赋值在过程式语言中无此限制。当该表达式翻译为编译后的文法中的一条赋值语句时，TAG 编译程序通过阻止对实现了该局部属性的已修改局部变量的访问，保障了值的稳定性。本例中，文法 G 最终综合出来的值要么是 n 的初始值（如果没有迭代），要么是对迭代运算中 $n@$ 的最终求值结果：

$$G \uparrow n \rightarrow [n = 0] (a \ [n@ = n + 1]) * b$$

10.3.3 向用户报告语法错误

帮助我们学习编译程序概念的大多数玩具编译程序往往迷失于实用的出错报告。本书从来没有省略对一个语法错误或约束错误的检测，但迄今为止，遇到这类错误的通常结果一直是以玉石俱焚方式终止编译过程，除报告“出现错误”之外没有任何其他的诊断信息。在一个将由编译程序编写人员之外的其他用户使用的实用编译程序中，这种做法是难以接受的。将出现的错误自动关联到失败的产生式并不是特别困难，但这些信息未必能给用户带来更多的启发；能够报告一个失败的可能原因的编译程序才会更有价值。

如第 4、7 章所述，研究文献中已有大量复杂的出错恢复和报告方法。我们倾向于一种更直接的方法，与之前让设计人员控制编译程序操作的做法保持一致。因而 TAG 编译程序有一种语法形式用于定义一条产生式规则失败时要报告的出错消息，而消息的内容则可由文法设计人员显

式地定义。报错消息由一个特殊的由方括号括住的语义动作序列组成，它们仅当直接跟在其后的语义动作或分析程序语法分析失败时才被求值。报错消息仅用于它所驻留的正规表达式形式中，因而一个迭代运算符的最后一次迭代可以（实际上也必须）在失败时不将错误向外传播到其外层的包围结构中，除非在该迭代运算符中指定了一个报错消息。类似地，在一个选择运算结构的选项体中的报错消息，只会在编译后的代码对该选项进行尝试并失败之后才被调用。

报错消息从左到右排序，因而语法中不同的语义动作或单词会引发不同的连续失败机会，每一种不同的情况均可用一个适当的报错消息序列作为关键码。因而，在以下语义动作序列中：

[断言 A；报错文本 B；断言 C；报错文本 D；断言 E]

断言 A 可静默地失败，不调用任何特定的报错消息；但断言 C 的失败将报告一个报错文本 B，断言 E 的失败将报告一个报错文本 D。如果将该序列置于一个选择运算的一个选项之中，或者它控制着一个迭代运算，则断言 A 的失败将直接前进到下一选项或终止迭代，而断言 C 或断言 E 的失败则将报告各自的报错消息，并终止整个编译程序的运行。这使得我们有可能构造语义制导报错消息，这些消息局部于待检查的约束。

TAG 编译程序中的一个报错消息规格说明在分隔语义动作的方括号中用一对双斜杠字符 (“// ... //”) 分隔。在报错消息的规格说明中不可访问非终结符，但可使用任何预定义的属性求值函数，包括引发错误或与其解释相关的属性的引用。报错消息中用引号括住的文本字符串，通常就是构成大部分屏幕显示出错消息文本的语义动作。

代码清单 10.3 中的文法片段演示了报错消息的用法。在该例子中，非终结符 **TypeClass** 返回一个数字，该数字反映了由树 **etype** 表示的类型种类。因而，乘法运算符 “*” 规定了类型种类 4、0 或 3（分别表示集合、整数或实数类型），但除法运算符拒绝整数类型，而整除运算符则要求整数类型。如果所有这些约束均未失败，则还须检查第二个操作数的类型也是兼容的，诸如此类。

代码清单 10.3 一个 TAG 文法片段中的报错信息

```
Term !env:table ^etype:tree ^evalu:tree ^iscon:bool

->  Factor !env ^etype ^evalu ^iscon
    TypeClass !env !etype ^tyc
    (("*" [// "Can't multiply that type"//]
      ([tyc = 4; opc = 11] | [tyc = 0; opc = 2] | [tyc = 3; opc = 7])
      | "/" [// "Can't divide that type"//]
      ([tyc = 4; opc = 12] | [tyc = 3; opc = 6])
      | ("REM" [opc = 5] | "DIV" [opc = 3] | "MOD" [opc = 4])
      [// "Whole number type required"//; tyc = 0]
      | ("AND" | "&") [opc = 11; // "AND requires BOOLEAN type"//]
      Lookup !env !-2 ^etype)
    Factor !env ^type2 ^valu2 ^iscon2
    [// "Expression term has incompatible types"//]
    Compatible !env !etype !type2
    ...
```

10.3.4 自动构造扫描程序

TAG 源文法的 **scanner** 部分定义了目标编译程序的部分扫描程序；扫描程序的其余部分

则隐式地定义为由引号括住的字符串文字量，它们就地分布在分析程序中的应有位置。TAG 编译程序为扫描程序中的这两类单词构造非确定的 FSA (NFA) 变迁，并将这些变迁收集到一个无序的列表中。分析了整个文法并将其转换为一棵中间树之后，一个树变换非终结符 **BuildScanner** 将第 3 章的算法应用到变迁列表以创建一个确定的扫描程序，然后将 FSA 的状态表翻译为可执行代码。

四类变迁按不同方式添加到 NFA 的列表中。其中两种是正则表达式结构产生的空变迁以及读字符变迁，构造 NFA 时通常需要这两种变迁。除空变迁和读字符变迁之外，还有语义动作空变迁和单词输出停机。停机状态有一个语义动作用于在停机时输出单个单词的编号，分析程序在向前看单词或接受动作中将用到这些单词编号。在转换为确定 FSA (DFSA) 的过程中，这些停机动作得以保留，然后被替换为等价的转入状态 0 的空变迁。当然，在最终完成的扫描程序中不会保留任何空变迁，因而每一单词变迁产生一个读变迁（仍附加着该单词的语义），这些读变迁从停机状态出发，转入由状态 0 出发、沿一个读变迁即可到达的每一状态；惟一例外是如果有从停机状态出发的读变迁，新变迁集中将删除这些输入符号。

图 10-3 演示了根据正则表达式

$$G \rightarrow a c [x] \mid b^+ [y]$$

构建的一个确定的 FSA。停机状态 H 输出单词 x ，停机状态 F 输出单词 y 。从状态 H 到起始状态 S 的空变迁立即被沿字符 a 转入状态 A、沿字符 b 转入状态 F 的变迁替代，这两个新变迁均保留了输出单词 x 的语义动作。类似地，停机状态 F 本来应有两个新变迁沿字符 a 和 b 出发，只是原本已有沿字符 b 出发的变迁，因而在新的变迁列表中可省略这一字符。现在容易看出，FSA 输出它的单词，然后继续读入下一单词，而不是停机。在实际应用中，扫描程序的过程 **Getoken** 将维护一个静态的状态变量，从而可将每一个输出单词返回给分析程序，并在再次被调用以读入下一单词时，可从上次停止执行的状态中恢复。

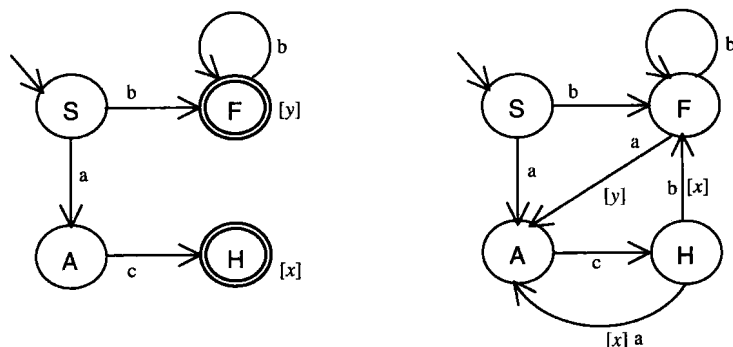


图 10-3 将停机状态转换为扫描程序的读变迁

语义动作变迁是书写在源文法中 **scanner** 部分的属性求值函数的结果。这些是空变迁，在构建一个确定的 FSA 之前必须消除它们，但语义动作代码却不能丢弃，而是在变换时将每一空变迁与其后继变迁合并，从而构造过程将语义动作从空变迁上转移到后续变迁上。这一效果类似于图 10-3 中的结果，但在构造过程的更早阶段执行变换。例如，如果从状态 A 到状态 B 有一个带语义动作 x 的空变迁，且从状态 B 到状态 C 和 D 有带语义动作 y 和 z 的变迁，则变换得到的 FSA 中，从状态 A 到状态 C 和 D 分别有带语义动作 xy 和 xz 的新变迁，并且原有的空

变迁被删除。先将空环路删除是很重要的，只有这样上述变换过程才会终止。

当扫描程序最终转换为可执行代码时，语义动作代码的每一字符串在状态表中均有一个入口；它们被指定了惟一的识别号，并且扫描程序代码中的 **CASE** 语句将根据每一状态的变迁选择合适的代码串来执行。扫描程序的构造过程一开始就指派了识别号，然后将识别号收集到集合中，以免在构造确定 FSA 的过程中因合并状态而产生不必要的代码副本。类似地，扫描程序的构造程序收集输入字母表中的字符放入等价类集合中，以压缩状态表的大小。代码清单 10.4 展示了 TAG 编译程序中扫描程序构造部分的结构，包括构造变迁列表并将该列表转换为代码的非终结符的框架。

代码清单 10.4 TAG 编译程序中，扫描程序的编译程序文法

```

TagCompiler                                     { 这是 TAG 的目标符号 }

→ ...                                           { 初始化和头部 }
"scanner"                                       { 开始处理扫描程序部分 }
(
  ID ↑name [newelt ↓tokset ↑tokno]             { 为它取一个单词编号 }
  DeriveAttrs ↓env ↓<> ↑attlist ↑nuenv          { 分析其属性列表 }
  "->" ScanRegExpn ↓nuenv ↓tranlist ↓tokno ↓0 ↑tostate ↑nutran
  [tranlist@ = <tk nutran <no>%tostate attlist>%tokno]
  [into ↓env ↓name ↓<tk attlist>%tokno ↑env@]   { 保存名字 }
  ";"
)+
"parser"                                       { 开始处理分析程序部分 }
...
"end" ID ↑tagname "."                          { 源文法结束 }
["MODULE "; spell ↓tagname; "; IMPORT Library;"]
                                           { 开始生成代码 }
BuildScanner ↓env ↓tranlist                   { 准备好输出扫描程序 }
thetree:BuildParser ↓env                      { 平展并输出分析和变换程序 }
... ;                                         { 结束、关闭文件... }

ScanRegExpn ↓env:table ↓intran:tree ↓tokn:int ↓fromst:int ↑tost:int ↑outran:tree

→ ScanAltern ↓env ↓intran ↓tokn ↓fromst ↑tost ↑outran
(
  "|" ScanAltern ↓env ↓outran ↓tokn ↓tost ↑ast ↑atran
  [outran@ = <tr atran <no>%ast <no>%tost <> <>>] { 空变迁 }
)* ;

ScanAltern ↓env:table ↓intran:tree ↓tokn:int ↓fromst:int ↑tost:int ↑outran:tree

→ [tost = fromst; outran = intran]             { 建立迭代程序 }
(
  ScanTerm ↓env ↓outran ↓tokn ↓tost ↑tost@ ↑outran@
)* ;

ScanTerm ↓env:table ↓intran:tree ↓tokn:int ↓fromst:int ↑tost:int ↑outran:tree

→ "("

```

```

[newstate ↓env ↑ast; newstate ↓env ↑tost]      { 两个新状态 }
[atrane = <tr intran <no>%fromst <no>%ast <> <>>] { 空变迁 ... }
ScanRegExpn ↓env ↓atrane ↓token ↓ast ↑tost ↑otran
)"
[xtran = <tr otrane <no>%ost <no>%tost <> <>>]
(
  [strane = <tr xtran <no>%ost <no>%tost <> <>>]
  [otrane = <tr strane <no>%ast <no>%tost <> <>>]
| "+"
  [otrane = <tr xtran <no>%ost <no>%ast <> <>>]
| "?"
  [otrane = <tr xtran <no>%ast <no>%tost <> <>>]
| [otherwise; otrane = xtran]
)

→ CHR ↑lochr [newstate ↓env ↑tost]              { 读变迁的新状态, }
[addset ↓lochr ↓empty ↑chset]                    { 沿所有这些字符出发 ... }
(
  "... CHR ↑hichr
  (
    [lochr < hichr; addset ↓lochr+1 ↓chset ↑chset@; lochr@ = lochr + 1]
  )*
)
[otrane = <tr intrane <no>%fromst <no>%tost chset <>>]

→ "["
  [newstate ↓env ↑tost]                          { 语义动作变迁的新状态 }
  SemanticAction ↓env ↓token ↑atree              { 构建语义动作树 }
]"
[otrane = <tr intrane <no>%fromst <no>%tost <> atree>] ;

ParseToken ↓env:table ↓intran:tree ↑otrane:tree ↑thetree:tree

→ STR ↑strno [newstate ↓env ↑tost]                { strno 是字符串表的索引 }
[chno = 0; xst = 0; atrane = intrane; length ↓strno ↑len]
{ 将串分解 }
(
  [chno < len; charfrom ↓strno ↓chno ↑achr] { 取一个字符串的字符 }
  [addset ↓achr ↓empty ↑chset]              { 沿该字符构建一条读变迁 }
  [atrane@ = <tr atrane <no>%xst <no>%ast chset <>>]
  [xst@ = ast; chno@ = chno + 1]
)*
[otrane = <tr atrane <no>%xst <>>%strno; thetree = <tk <>>%strno] ;

BuildScanner ↓env:table ↓tranlist:tree

→ tranlist:ScannerLists ↑thevars ↑theActs ↑chrsets
{ 提取集合和列表 }
thevars:VarList ↓true ↓true                { 输出扫描程序的 VAR 声明 }
tranlist:FindCycle ↓0 ↓0                    { 删除空环路 }
tranlist:EmptyMoves                          { 删除空变迁 }
[addset ↓0 ↓empty ↑stsets; nutrans = <>]   { 转换为确定的 FSA }
(

```

```

    AnotherSet ↓stsets ↑stno                { 处理下一状态, 或失败 }
    TransSet ↓tranlist ↓nutrans ↓stsets ↓stno ↑stsets@ ↑nutran@
  ) *
  nutran:HaltMoves                          { 将停机状态转换为变迁 }
  nutran:OutStatenCharTables ↓chrsets       { 输出已完成的确定 FSM 表 }
  ["PROCEDURE Getoken; BEGIN REPEAT"]       { 正确地输出扫描程序 }
  ["ReadChar(inch); temp := ChTable[inch]; CASE StTable[state, temp] OF"]
  theActs:Flatten ↓env                      { 输出语义动作的代码 }
  ["END; state := StTable[state, temp + 1] UNTIL NexToken # 0 END Getoken;"]
  tranlist:OutTokProcs ↓env ;               { 执行带属性的单词过程 }

```

代码清单 10.4 的 TAG 编译程序文法中, 分析扫描程序规格说明的部分相当简单。正则表达式被分析并转换为由空变迁连接的读字符变迁。停机状态 (**tk** 结点) 被添加到变迁列表中, 稍后由 **BuildScanner** 处理。编译 TAG 中的 **parser** 部分时, 非终结符 **ParseToken** 负责识别字符串文字量, 将它们分解为各自的一个个字符, 并构建一个从状态 0 开始、以另一停机状态结束的变迁序列。

BuildScanner 的大部分编译时间花费在迭代运算符根据消除空变迁后的变迁列表构造一个确定的 FSA。迭代运算符第 1 行的非终结符 **AnotherSet** 查找未被转换到新 FSA 中的状态集, 或在不再有这些状态时停止迭代过程。第 2 行的非终结符 **TransSet** 查找从 **stno** 指定的集合中的所有状态出发的变迁, 对约简后的输入字母表中的每一符号, 创建一个状态集添加到列表 **stsets** 中 (并返回更新后的列表)。新的状态机还扩展了一个变迁, 沿该符号转入新的状态集编号。

为标准 Modula-2 编译程序构造的扫描程序采用完全解码的保留字, 它的 FSA 有大约 220 个状态, 其中输入字母表约简为 57 个等价类和大约 150 个不同的语义动作序列 (包括输出的所有单词)。将 FSA 的状态约简为最小数目仅可消除不足 20 个状态, 可谓得不偿失, 故 TAG 编译程序省略了这一过程。与编译程序的其他部分相比, 最后得到的状态表的大小相当合理。如果将保留字作为标识符扫描并从字符串表中取出, 而不是由 FSA 识别, 则状态数目还可约简 80%, 并且输入字母表和语义动作集的大小可减半, 因而直接编写代码实现 FSA 可能更加切实可行。

然而, 除上述特点之外, 选择直接执行状态机并不会优于基于表格的实现, 即便采用了一个完美散列函数将保留字的识别简化为字符串表的单个比较。还应留意到, 为一个给定的保留字集合找出一个完美散列函数所花的时间远多于构建一个状态机识别程序。TAG 编译程序约有一半时间花费在将 FSA 转换为确定的扫描程序; 另一半时间几乎平均花费在分析源文法, 以及生成输出的 Modula-2 程序文本。

10.3.5 TAG 编译程序的语法分析

与 **scanner** 部分相比, 将一个 TAG 源文法剩余的两部分编译为输出代码的方法相当平淡, 基本遵循第 4、5 章介绍的过程。这些部分被分析并转换为一种中间树结构, 这一树结构记录了为一个扩展了正则表达式的上下文无关文法的抽象语法。非终结符的向后引用充分利用了已可用的属性类型信息设置由这些引用综合出来的局部属性的类型; 非终结符的向前引用中的类型检查 (参数匹配) 推迟到代码输出阶段才执行。

在分析源文法的同时, 执行少量数据流分析即可确定一个属性断言到底是一个应求值为 **true** 或 **false** 的断言, 还是一个应编译为一条赋值语句的属性求值函数; 这些数据流分析比

类型检查稍复杂一些。该分析工作利用了一对集合，其中一个集合由所有已被赋值的标识符组成，另一集合由可能被赋值、也可能不被赋值的标识符组成。如果一个标识符的首次出现是作为一个非终结符引用返回的综合属性，或该标识符与一个已定义的属性表达式比较是否相等，则将该标识符同时添加到两个集合中。对该标识符的后续引用会在已赋值标识符的集合中找到该标识符，因而这些引用将被编译为等式断言，该断言的失败会异常终止其代码片段的执行。当两个或多个选项在闭圆括号处重新汇合时，已赋值标识符的集合被每一支路的集合的交集替代；第二个集合则通过并运算合并其分量。类似的合并将星号表示的迭代运算的结果与它的输送集合组合在一起。如果后续引用的一个标识符在第二个集合中、但不在第一个集合中，则 TAG 编译程序将报告一个错误。标识符的另一集合类似地追踪了迭代运算的属性。

TAG 编译程序为变换程序以及分析程序中不读入输入单词的非终结符创建了一个不确定的递归下降分析程序。这使得语义驱动的分析程序构造过程能够处理任意复杂的属性文法，特别是编译程序设计人员可在语法上书写仅检查语义性质的空非终结符，然后在该非终结符被调用的所有地方为调用结果构建一个选项。如果一个空非终结符的语义断言检测失败，且没有明确的报错消息，则执行过程从该非终结符退回，并寻找另一选项。如果语义断言检测失败之前已读入了一个单词，或读单词本身失败，或该失败定义了一个明确的报错消息，则报告一个语法错误并终止编译过程。

根据文法编译得到的代码有两种方法可在断言或非终结符引用失败时，从一个迭代、选择或非终结符规则中跳出并退回。最简单且最直接的方法是生成一条条件分支语句或 GOTO 语句，在每一潜在的失败之后立即跳到下一选项或适当的出口。在 Modula-2 这类高级语言中，上述实现通常可编码为一系列嵌套的 IF 子句，但这样的编码有两个问题。最严重的问题是高度结构化的代码使得如果第一个选项的第二个断言失败时，难以调用第二个选项；特别是考虑到在第一个断言失败时，第二个断言甚至不应被求值的情况。Modula-2 语言的条件语句中布尔测试采用显式的短路求值，因而这样的代码只是笨拙一些，但并不是不可能的。第二个问题是许多 Modula-2 语言的实现强制规定了可编译的嵌套语句结构的层数，而一些合理的文法可能会产生数百层子句深度的结构。

TAG 编译程序分配了一个布尔变量用于保存当前成功的状态，以较小的性能代价避免了上述两个问题。在每一可能导致失败的语句之后的代码周围，都对该变量进行简单的测试。由于每一测试都是直接包围的，故对该标志的测试次数比基于 GOTO 语句的实现中的测试更多，但这对于序列结构或嵌套结构是没有问题的。此外，具有良好优化技术的编译程序（本书毕竟是一本关于编译程序设计的教材）可应用分支链窥孔优化技术，生成恰好与嵌套条件语句同样（最优）的代码。

代码清单 10.5 展示了 TAG 编译程序中代码生成部分的最重要内容。只要充分参阅附录 B，读者自己构造一个编译程序前端应该不会遇到什么困难；该前端构建的树将由变换程序平展为代码。这一实验项目留作本章的一个练习。

代码清单 10.5 TAG 编译程序中代码生成程序的文法

```
BuildParser ↓env:table
```

```
→ <nt link inh der atts body>%name
```

```
{ 一个非终结符的子树 }
```

["PROCEDURE "; spell ↓name; "("]	{ 生成过程的头部 }
inh:VarList ↓false ↓(der # <>)	{ 继承属性 }
der:VarList ↓true ↓false	{ 综合属性 }
["): BOOLEAN; FORWARD;"]	
link:BuildParser ↓env	{ 处理其他头部 }
["PROCEDURE "; spell ↓name; "("]	{ 再次生成过程的头部 }
inh:VarList ↓false ↓(der # <>)	{ 继承属性 }
der:VarList ↓true ↓false	{ 综合属性 }
["): BOOLEAN; VAR ok: BOOLEAN;"]	
atts:VarList ↓false ↓true	{ 输出局部属性的声明 }
["BEGIN ok := TRUE;"]	
body:Flatten ↓env	{ 将过程体树平展为代码 }
["; RETURN ok END"; spell ↓name; ";"]	{ 过程结尾 }
→ <> ;	{ 列表的结尾 }
Flatten ↓env:table	{ 生成一条语句 }
→ <ca left right>	{ 连接运算的结点 ... }
left:Flatten ↓env	{ 平展左子树 }
["IF ok THEN"]	
right:Flatten ↓env	{ 平展右子树 }
["END;"]	
→ <al left right>	{ 选择运算的结点 ... }
left:Flatten ↓env	{ 平展左子树 }
["IF NOT ok THEN ok := TRUE;"]	
right:Flatten ↓env	{ 平展右子树 }
["END;"]	
→ <st body>	{ 闭包运算的结点 ... }
["WHILE ok DO"]	
body:Flatten ↓env	{ 平展迭代体的子树 }
["END; ok := TRUE;"]	
→ <tk <>>%tokn	{ 字符串文字量单词引用 }
["ok := MatchToken("; number ↓tokn; ");"]	
→ <tk parms:<at ...>%tokn	{ 带属性的单词引用 ... }
["ok := MatchTok"; number ↓tokn; "("]	
parms:DoArgs ↓env ↓true ↓false ↓<> ↑xx ↑post	
["];"]	
post:Flatten ↓env	{ 平展后置断言 }
→ <nt send recv>%name	{ 非终结符的引用 ... }
["ok := nt"; spell ↓name; "("]	
send:DoArgs ↓env ↓false ↓<> ↑coma ↑postx	
recv:DoArgs ↓env ↓coma ↓postx ↑xx ↑post	
["];"]	
post:Flatten ↓env	{ 平展后置断言 }
→ <cs link body>%num	{ 选择运算的情况选择符 ... }
link:Flatten ↓env	{ 平展表的剩余部分 }

```

    [" | "; number ↓num; ": "]
    body:Flatten ↓env                                { 平展选择体的子树 }

→   <as expt thev>                                    { 赋值语句的结点 ... }
    thev:GetVarType ↓env ↑tipe
    thev:ShoVar
    [" := "]
    expt:Flattex ↓env ↓tipe ↑xx                      { 平展表达式的子树 }
    [";"]

→   <mt expt>                                          { 断言的结点 ... }
    ["ok := "]
    expt:Flattex ↓env ↓2 ↑xx                          { 平展表达式的子树 }
    [";"]

→   <> ;                                              { 无代码 }

ShoType ↓needsem:bool

→   <ty>%tipe
    ([tipe = 1; ": INTEGER"] | [tipe = 2; ": BOOLEAN"] | [tipe = 3; ": Tree"])
    ([needsem = true; ";"])?
    ;

VarList ↓needvar:bool ↓needsem:bool

→   ident:<at link tipe> | ident:<vr link tipe>
    link:VarList ↓needvar ↓true                        { 按列表的逆序执行 }
    ([needvar = true; "VAR"])?
    ident:ShoVar
    tipe:ShoType ↓needsem

→   <> ;                                              { 列表的结尾 }

ShoVar

→   <at ...>%name                                     { 继承属性或综合属性 }
    ["at"; spell ↓name]

→   <vr ...>%name                                     { 局部属性 }
    ["vr"; spell ↓name]

→   <tv ...>%varno                                    { 编译程序生成的临时变量 }
    ["tv"; number ↓varno] ;

Flattex ↓env:table ↓mustype:int ↑typex:int            { 为表达式的树生成代码 }

→   thev:<vr ...> | thev:<at ...> | thev:<tv ...> | { 属性或临时变量的引用 }
    thev:GetVarType ↓env ↑typex
    thev:ShoVar
    [// "Attribute type mismatch"//; mustype * (mustype - typex) = 0]

→   <cn <ty>%l>%num                                  { 整数常量 }

```

```

[number ↓num; //"This is Integer"//; mustype < 2; typex = 1]

→ <cn <ty>%3>                                     { 空树常量 }
   ["NIL"; //"This is a tree type"//; mustype * (mustype - 3) = 0; typex = 3]

→ <cn <ty>%2>%0                                     { 布尔常量 ... }
   ["FALSE"; //"This is Boolean"//; mustype * (mustype - 2) = 0; typex = 2]

→ <cn <ty>%2>%1                                     { 布尔常量 ... }
   ["TRUE"; //"This is Boolean"//; mustype * (mustype - 2) = 0; typex = 2]

→ <ng expt>                                         { 一元求负结点 ... }
   ["-("]
       expt:Flattex ↓env ↓1 ↑xx                     { 平展子树 }
   [")"; //"This type is Integer"//; mustype < 2; typex = 1]

→ <ad left right>                                   { 加法结点 ... }
   ["("]
       left:Flattex ↓env ↓1 ↑tipe                    { 平展左子树 }
   [") + ("]
       right:Flattex ↓env ↓tipe ↑xx                  { 平展右子树 }
   [")"; //"This type is Integer"//; mustype < 2; typex = 1]

→ <mp left right>                                   { 乘法结点 ... }
   ["("]
       left:Flattex ↓env ↓1 ↑tipe                    { 平展左子树 }
   [") * ("]
       right:Flattex ↓env ↓tipe ↑xx                  { 平展右子树 }
   [")"; //"This type is Integer"//; mustype < 2; typex = 1]

→ <eq left right>                                   { 判断是否相等的结点 ... }
   ["("]
       left:Flattex ↓env ↓0 ↑tipe                    { 平展左子树 }
   [") = ("]
       right:Flattex ↓env ↓tipe ↑xx                  { 平展右子树 }
   [")"; //"This type is Boolean"//; mustype * (mustype - 2) = 0; typex = 2]

→ ...                                               { 其他运算符 ... }

DoArgs ↓env:table ↓incoma:bool ↓inpost:tree ↑outcoma:bool ↑outpost:tree

→ <ax link expt post>%tipe                           { 生成过程参数的代码 }
   link:DoArgs ↓env ↓incoma ↓inpost ↑coma ↑postx
   expt:Flattex ↓env ↓tipe ↑xx                       { 平展表达式的子树 }
   ([post = <>; outpost = postx] | [otherwise; outpost = <ca postx post>])
   ([coma = true; ", "])?
   [outcoma = true]

→ <> [outcoma = incoma; outpost = inpost] ; { 表的结尾 }

```

代码清单 10.5 中的非终结符 **DoArgs** 需要解释一下。它用于将一个过程调用中的实参表平展为 Modula-2 语言的表达式和变量引用。对于一个继承属性而言，结果恰好就是平展后的表

达式；对于一个综合属性而言，分析程序早已确定了它是一个局部属性的定义引用，还是一个限定该综合属性与某一表达式的值相等的断言。如果它定义了一个值，则表达式树仅由局部变量或参数的名字组成；否则，表达式树是一个指向编译程序生成的某一临时变量的引用，并且后置断言子树中包含了判断它与文法中指定的表达式相等的断言代码。由于一个过程调用的参数按这种方式平展，后置断言汇集到一棵树中；在过程调用返回之后，单独将该树平展并求值。每一后置断言通常具有以下形式：

```
<mt <eq <tv>%123 expt>>
```

这一断言要求指定的临时变量（此处的数字 123）的值必须与指定表达式的值相等。它被平展为一条 Modula-2 语句后，形如：

```
IF ok THEN  
  ok := (tv123) = (expnvalue);  
END;
```

代码清单 10.5 中的文法省略了 TAG 编译程序中代码生成的两个重要方面：树变换和出错报告。这两方面分开来会更容易理解。

10.3.6 树变换

树变换包括三个步骤：将源树与文法模板匹配（分析源程序）、构造替代树以及用替代树替换源树。TAG 编译程序中使用了三个库例程支持这些功能。替换过程并不是特别困难，但有必要将源树根结点的内容替换为相应的替代树根结点的内容，从而对源树的其他引用将继续引用替代树。编译得到的树变换非终结符的最后一句是调用执行这一替换的库例程。

树的构造可出现在属性求值表达式中，分析程序可利用该表达式构建原中间代码树；树的构造还可出现在一个变换非终结符中指定一棵替代树时。构造过程借助于单个库函数完成，该函数根据分量子树构建一个结点。作为返回一个树结点的函数，它可嵌套在一个由源文法的树模板结构所支配的结构中，这简化了对它的编译。在编译后的编译程序中，一条赋值语句可构建一棵任意复杂的子树。假设所有结点最多只有四棵子树（用 NIL 表示无用的子树即可构建更小的结点），该库例程具有以下接口：

```
PROCEDURE Build(noden, nsubs: CARDINAL; decor: Tree;  
  s1, s2, s3, s4: Tree): Tree;
```

树变换文法中最困难的部分是识别一棵给定的树何时与文法中指定的模板匹配。模板中待匹配的每一结点必须与源树中对应结点的结点名和子树数目均相同（那些用符号“..”指定为不关心的子树除外）。模板中用一个标识符命名了的子树应赋值给同名的局部变量或参数，不管该名字是否作为附加模板结构的标签。

TAG 可能有两种方法解决上述识别问题。最直接的方法是在分析该结构中的每一结点时，将指向该结点的指针赋值给一个临时局部变量，然后在分析时析取该变量，以取出其内容部分。带名字的子树结点被赋值给指定的局部变量，而不是赋值给一个临时变量。在向下深入到其内部结构之前，必须检查每一结点的结构，以避免析取不存在的子树指针。类似于从一个失败的选项中跳出，这里也出现了编译后的高级语言代码结构问题；这虽是一个小问题，但却是不可忽略的问题。

另一种方法要求使用一个或多个布尔类型的库过程，用于测试源树的各个细节。为此 TAG

编译程序利用了单个函数 **TreePart**，如代码清单 10.6 所示。该函数接受的参数包括一个指针（指向源树的根结点）和一个数字（该数字编码了树结构中从根结点到待测试细节的特定路径）；如果目标树中存在这一完整的路径，且指定的细节能够匹配，则返回 **true**，否则返回 **false**。由于这是一个布尔函数，将每一细节的测试结果连接起来（使用 **AND** 运算符），就可在单条条件语句测试中匹配整个结构。**Modula-2** 语言的布尔表达式短路求值特性保证了只要有一个细节匹配失败，整个测试将异常终止。如果该库例程将一条成功路径及其中间指针缓存在局部静态存储器中，则对深层嵌套的相邻细节进行连续测试时，可充分利用先前调用中已完成的路径验证和指针析取，从而可进一步提升性能。**TAG** 编译程序也可能保存中间子树的指针（但并没有这样做），并基于这些指针开始额外的测试，而不是每一次都从根结点开始测试。

代码清单 10.6 分析树模板中一个细节的库例程

```

TYPE
  Tree = POINTER TO Node;
  Node = RECORD
    nodename: INTEGER;
    decor: Tree;
    subtrees: ARRAY [1 .. 6] OF Tree
  END;

PROCEDURE TreePart (TreePtr: Tree; Pathop, CompareTo: INTEGER; VAR ReTree: Tree): BOOLEAN;
(* Pathop 的含义:
  1-6: 选择一棵子树并继续
  7: 返回一个装饰
  8: 如果结点名等于 CompareTo 则返回 TRUE, 否则返回 FALSE
*)
VAR
  index: INTEGER;
  result: BOOLEAN;
BEGIN
  ReTree := TreePtr;
  result := TRUE;
  WHILE result AND (Pathop > 7) AND (ReTree # NIL) DO
    WITH ReTree^ DO
      index := Pathop MOD 8;
      Pathop := Pathop DIV 8;
      IF (index = 0) OR (index = 7) THEN
        result := FALSE
      ELSE
        ReTree := subtrees[index]
      END
    END (* WITH *)
  END; (* WHILE *)
  IF result THEN
    IF ReTree = NIL THEN
      RETURN (Pathop = 0) AND (CompareTo < 0)
    ELSIF Pathop = 0 THEN
      RETURN ReTree^.nodename = CompareTo
    ELSIF Pathop = 7 THEN
      ReTree := ReTree^.decor
    ELSE

```

```

    ReTree := ReTree^.subtrees[Pathop]
  END (* IF ReTree *)
END; (* IF result *)
RETURN result
END TreePart;

```

代码清单 10.6 中的库例程将路径编码为一个整数，该整数划分成长度为 3 个位的子段，每一子段指明了一个路径片段。长度不超过 10 段的路径有可能用宿主计算机中的 32 位整数表示，但在文法源文本中嵌套了 3 层或 4 层结点深度的树模板则未必适合这样的表示，因为这样会导致可读性较差。长度为 3 位的段可表示的值允许选择一个结点名以及子树数目、一个装饰、或不超过 6 棵的子树；必要时，可代之以一个字符串常量或有更多位的另一数据类型，从而扩展其表示范围。TAG 编译程序版本中的 **TreePart** 利用一个 **VAR** 参数返回指定路径中最后的子树，而在另一个参数中接受待匹配的结点名的值。

分析程序负责判断源文法中的一个树模板是待匹配的还是待构造的，并构建一个适当的中间树以表示模板的结构。当对匹配的分析每次递归携带一个继承属性向下传递时，递归过程沿树模板下降；该继承属性可构建对一条路径进行编码的常量。因而，中间代码树已包含了在一个模板匹配条件语句中生成每个单独的细节测试所需的所有信息。

10.3.7 语法错误停机

在编译后的编译程序中，令人满意的出错报告会对源文法中每一文法形式必须生成的代码类别产生深远的影响。TAG 源语言提供了一种语法形式用于指示错误，但这一语法形式仅适用于定义它的结构层次中。

考虑如下文法片段：

```

[// "ErrorMessage1" //]
DoSomething
(ThisOrThat | [// "ErrorMessage2" //] OnTheOtherHand)
(RepeatSomething)*
SomethingElse

```

如果 **DoSomething** 或 **SomethingElse** 失败，将报告 **ErrorMessage1**；但如果 **ThisOrThat** 失败，就不会有报错消息，除非 **OnTheOtherHand** 也失败，此时会报告 **ErrorMessage2** 而不是 **ErrorMessage1**。如果 **RepeatSomething** 失败，也不会有任何报错消息，因为迭代体的失败是终止迭代的惟一方法。还应注意，如果其中一个非终结符包含其自身的报错消息规格说明，则它们在失败时将使用这些报错消息，而不是此处指明的消息。当然，此处展示的非终结符可用表示局部语义动作的方括号替代，而报错消息仍可以同样的方式应用；这也是理应如此。

有两种方法可让一个出错声明仅在局部起作用。编译程序可追踪当前报错消息声明的作用域，并在每次错误测试时找到正确的报错消息。编译程序也可定义一个变量用于保存一个指明报错消息的索引，然后在每次进入一个新结构时将该变量压入一个栈中，并且在退出该结构时再将该变量从栈中弹出。基于栈的方法的代码更慢一些，但稍微更紧凑一些，并且相同的栈还可用于实现编译程序的其他需求（参阅练习 8）。

小结

本章考虑了 Lisp 这一类应用式程序设计语言的编译程序设计问题，利用 Scheme 语言的术语介绍了

与应用式语言相关的编译程序设计问题, Scheme 语言是表处理语言 Lisp 的一个派生语言。作为编译程序设计的一部分, 本章讨论了与编译一个可使用的程序有关的一些问题。本章给出的两个 Tiny Scheme 语言文法分别对源程序文本进行分析并转换为表, 以及将表编译为 Itty Bitty 栈机器代码。

本书以属性文法贯穿了整个编译程序设计过程, 最后描述了基于 TAG 的编译程序自动构造。TAG 给出了一个程序 (即编译程序) 的规格说明, 本章介绍的 TAG 编译程序是一个自编译的 TAG, 它将一个变换属性文法转换为标准的 Modula-2 代码。

关键术语

applicative language (应用式语言) 以函数应用为惟一控制结构的程序设计语言。

atom (原子) Lisp 语言中的一个原子数据 (名字或数字), 并不是一个点对。

call/cc 在 Scheme 语言中表示以当前延拓调用 (call-with-current-continuation 的缩写)。

car Lisp 或 Scheme 语言中的一个函数, 提取一个点对的左子树, 或一个表的第一个元素。习惯上用于表示一个点对的左部。

cdr Lisp 或 Scheme 语言中的一个函数, 提取一个点对的右子树, 或一个表剔除第一个元素后的剩余部分。习惯上用于表示一个点对的右部。

cons Lisp 或 Scheme 语言中的一个函数, 根据两个值构造一个点对, 这两个值分别作为点对的左、右子树。如果右子树是一个表, cons 扩展该表的方法是将左子树插入到右子树的前面。

continuation (延拓) 一个部分求值函数, 在 Scheme 语言中可用 call/cc 捕获。

curry (柯里化) 将一个含多个参数的函数转换为一个比原来参数数目更少的函数, 返回的另一个函数可应用到 (一个或多个) 剩余的参数。

dotted pair (点对) Lisp 语言中的基本数据结构。

dynamic scoping (动态作用域) 自由变量在调用环境中求值。

free variable (自由变量) 非局部的变量。

Lisp (Lisp 语言) J. McCarthy 于 1958 年提出的一种应用式程序设计语言。Lisp 表示表处理 (List processing), 常用于人工智能研究中。

Scheme (Scheme 语言) G. Sussman 和 G. Steele 于 1975 年提出的一种 Lisp 语言变种。关于 Scheme 语言的简介还可参阅 [Smith, 1988]。

static scoping (静态作用域) 一个自由变量的含义取决于函数声明周围的程序文本, 由此确定对该变量求值的上下文。

tail-recursion (尾递归) 如果递归调用所计算的值在递归展开时不加修改就向上传递, 则称该函数是尾递归的。

练习

1. 给出代码清单 10.2 针对以下程序产生的分析树表示:

(a) (lambda (x y) (cons y x))

(b) (cons x (cons x (cons y (cons y z))))

2. 给出等价于以下 Scheme 函数的尾递归函数:

(a) (define power (lambda (x y)

(if (= y 0) 1 (* x (power x (- y 1))))))

(b) (define Fibonacci (lambda (n)

(if (< n 2) 1 (+ (Fibonacci (- n 1)) (Fibonacci (- n 2))))))

3. 将练习 2 得到的函数转换为传统的循环 (迭代) 形式。

4. 讨论练习 3 中过程的优化。不要忘记在你的讨论中包括所有新的或所需的过程。

5. 给出由以下表达式构建的确定有穷状态自动机 (DFSA)。
 - (a) $G = aca[x] \mid b^+a[y]$
 - (b) $H = a[x] \mid ba^+c[y] \mid a^+bc^+[z]$
6. 将练习 5 得到的确定有穷状态自动机中的停机状态转换为扫描程序的读变迁。
7. 为 IBSM 的 Scheme 编译程序编写一个库例程, 该例程将任意数目的参数复制到栈偏移量位置之下指定数目的字 (这些参数将取代调用者自己的参数); 然后为代码清单 10.2 中的文法添加一个非终结符 **TailCall**, 该非终结符将一个尾递归编译为对上述例程的调用。
8. 为非终结符 **Flatten** 添加树平展的代码, 使得它为 **Plus** 迭代运算符生成正确的代码。注意 **Plus** 迭代运算要求在因一个失败的断言检测而跳出循环之前, 至少有一次完整的迭代 (无内部失败) 被视为成功的。证明你的实现同样可正确地处理嵌套的迭代。在实现时, 还应注意在栈中存放单个循环控制变量与分配多个临时变量 (需要一个随信息流动的属性对它们进行计数) 这两种设计之间的折中。

复习小测验

指出下列陈述是否正确。

1. 应用式语言允许对变量赋值, 但不允许函数调用。
2. Lisp 语言中的算术运算采用前缀 (波兰式) 表示法书写。
3. 若要处理由 λ 表达式创建的函数, 则 Lisp 编译程序必须在运行时是可用的。
4. 将 **cons** 函数应用到参数表, 就可以很容易地编译尾递归。
5. 针对变换程序以及那些读入输入单词的分析程序非终结符, TAG 编译程序创建一个不确定的递归下降分析程序。
6. TAG 是一种数据流语言。
7. TAG 的 **transformer** 部分在一个上下文无关的树变换属性文法中定义了待编译的编译程序的代码优化和代码生成需求。
8. 一个 TAG 恰好有两个语法组成部分: **scanner** 和 **parser**。
9. TAG 是一种非过程式的程序设计语言。
10. 函数 **f(x)** 无需柯里化。

进一步阅读

- Curry, H.B. & Feys, R. *Combinatory Logic*, Vol. 1. Amsterdam: North-Holland, 1968.
- Ganzinger, H., Ripken, K., & Wilhelm, R. "Automatic Generation of Optimizing Multipass Compilers." *Proceedings of the IFIP 1977 Congress*, American North-Holland (1977), pp.535-540.
- Halstead, R.H. "Multilisp: A Language for Concurrent Symbolic Computation." *ACM Transactions on Programming Languages and Systems*, Vol.7, No.4 (October 1985), pp.501-538.
- 使用 **pcall** 结构扩展了 Scheme 语言的并行执行 (例如, **pcall cons x y** 将并发地对 **x** 和 **y** 求值)。
- Kessler, P.R. et al. "EPIC: A Retargetable, Highly Optimizing Lisp Compiler." *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, Vol.21, No.7 (July 1986), pp.118-130.
- 介绍了一个“7 遍”Lisp 编译程序 (详细地给出了每一遍的描述), 可作为不同 Lisp 编译策略的试验台。
- Leverett, B.W. "An Overview of the Production Quality Compiler Compiler Project." Carnegie-Mellon, 1979.
- 关于 BLISS-11 编译程序的优化建议的非正式汇集。
- McCarthy, J. "History of Lisp." *History of Programming Languages*, NY: Academic Press, 1981, pp.173-197.
- Scott, D.S. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, Cambridge, MA: MIT Press, 1987.
- 参阅第 40~41 页关于柯里化函数的通俗解说。

Schönfinkel, M. "Über die Bausteine der mathematischen Logik." *Mathematische Annalen*, Vol.92 (1924), pp.305-316.

Smith, J.D. *An Introduction to Scheme*, Englewood Cliffs, NJ: Prentice Hall, 1988.

参阅第 16.2 小节（特别是第 223~226 页）关于 Scheme 语言中延拓的解释。

Summers, P.D. "A Methodology for Lisp Program Construction from Examples." *Journal of the ACM*, Vol.24, No.1 (January 1977), pp.161-175.

通过为若干输入—输出样本构造谓词和程序片段，根据样本规格说明导出递归的 Lisp 程序。

Sussman, G.J. & Steele, G.L. *Scheme: An Interpreter for Extended Lambda Calculus*. MIT Artificial Intelligence Memo, 349, 12 (1975).

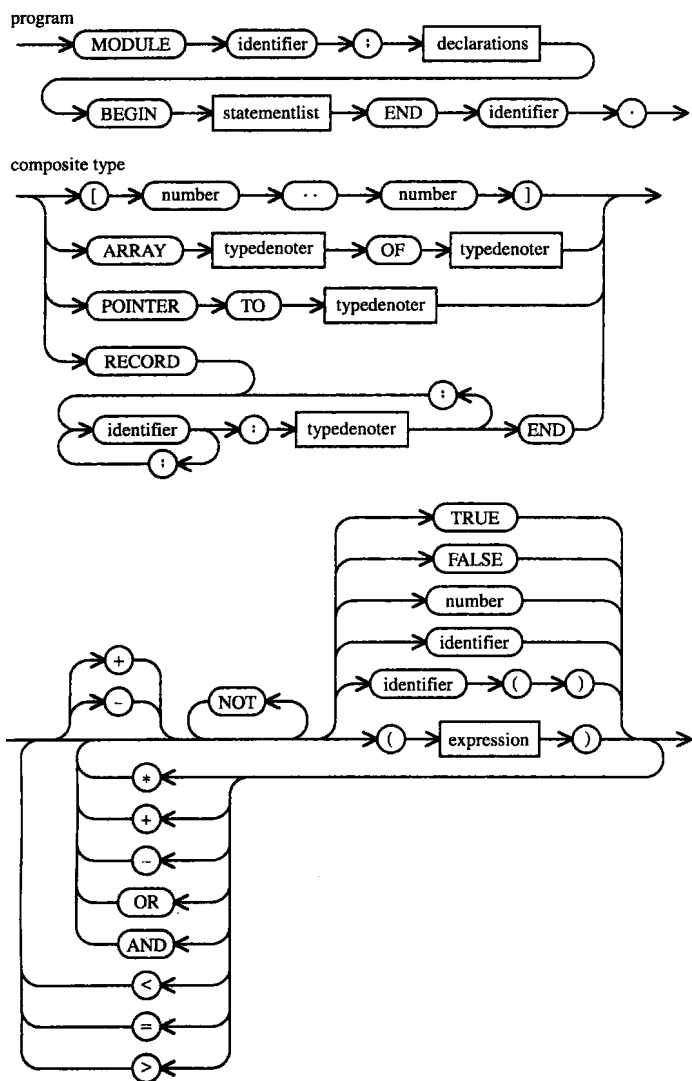
Waters, R.C. "User Format Control in a Lisp Prettyprinter." *ACM Transactions on Programming Languages and Systems*, Vol.5, No.4 (October 1983), pp.513-531.

这一采用 Lisp 语言编写的美化打印工具提供了一种通用机制，既可用于打印数据结构，也可用于打印程序。

Waters, R.C. "Efficient Interpretation of Synchronizable Series Expressions." *Proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques* (June 1987), pp.74-85.

Wise, D.S. "Functional Programming." *Encyclopedia of Computer Science and Engineering* ed. A. Ralston & E.D. Reilly. New York: Van Nostrand Reinhold, 1983, pp.647-650.

附录 A Itty Bitty Modula 语法图



备注：

1. 尽管 Modula-2 语言标准要求在文法的某处有一个标识符，但在语法上可允许任意的类型指示符，并且仍可拒绝错误的程序。约束程序应拒绝此处匿名声明的所有复合类型，因为该函数的值不可能组成类型正确的用法。

2. 标识符以一个字母开头，并可包含字母和数字；标识符的大小写是有意义的。

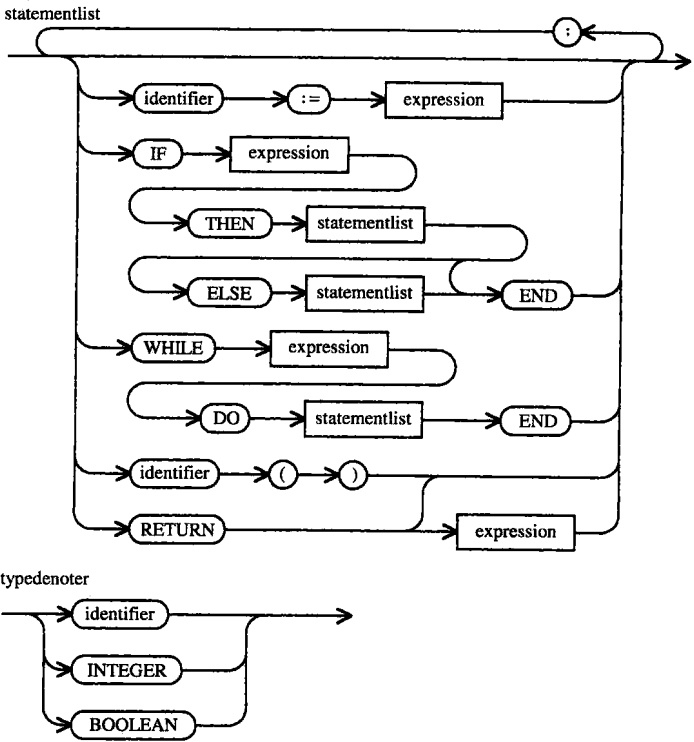
3. 运算符的优先级为：

()

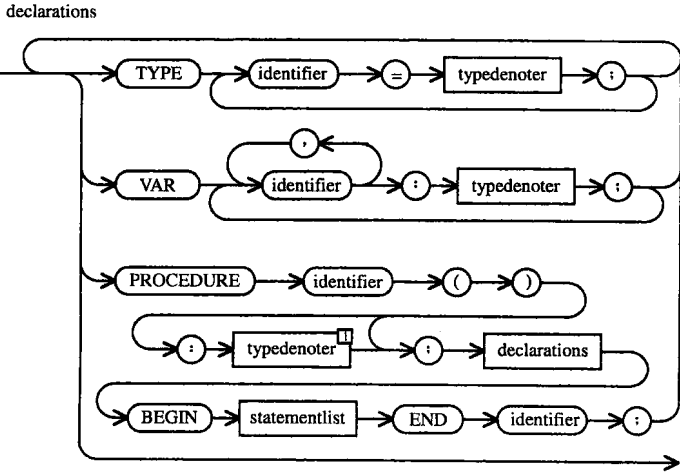
NOT

* AND

+ - OR
< = >



- 4. 尽管本附录的语法图中未展示出来，关系运算符并没有结合性。从语法上很容易强制规定这一点。
- 5. 尽管本附录的语法图允许，但 “- NOT a” 是非法的。通常可认为这在语法上是正确的，但在约束程序中可阻止这一用法。
- 6. 注释和空白通常由扫描程序删除。Itty Bitty Modula 语言的注释以 “(* ” 开头、以 “ *) ” 结尾，其中可包含任意字符，包含 “(”、“) ” 和 “ * ”，但不可包含 “(* ” 或 “ *) ”。



附录 B TAG 编译程序的 TAG

尽管 TAG 编译程序作为一种研究工具仍在不断演化中，但它依然既可作为变换属性文法 (TAG) 的形式化定义，也可作为将 TAG 翻译为 Modula-2 语言的一个编译程序源代码。此处的代码清单定义了 TAG 的完整语法，并给出了足够的语义使得读者可构建一个可用的 TAG 编译程序。它还可作为一个良构的 TAG 实例。

代码清单 B.1 TAG 编译程序的 TAG^①

```
tag TagGrammar:

{tree node names}
type node(no, ty, gl, fn, tk, nt, st, pl, qu, tr, ac, at, mt, ot, cn, rt,
{16}      xf, bt, er, al, ca, qm, dl, rp, dr, vr, as, eq, ls, gr, ne,
{31}      an, or, ad, su, mp, dv, md, ng, tv, rn, nn, nc, co, sa);

{ node names:
  <ac -- not used                <dr -- not used
  <al alternation                <eq equate
  <at param defn, ref           <ls less than
  <bt build tree                <rp -- not used
  <ca concatenate              <tr scanner transition
  <cn constant                  <ty type
  <dl dollar iteration          <vr local variable
  <er error code                <xf transform tree
  <fn function defn, call       <gr greater
  <gl global var                <ne not equal
  <mt "must true" asserts its subtree <an and
  <no not, also data structure  <or or
  <nt nonterminal defn, call    <ad add
  <ot output text               <su subtract
  <pl plus iteration            <mp multiply
  <pl plus op                   <dv divide
  <qm -- not used               <md mod
  <qu -- not used               <ng negative
  <rt recognize subtree         <tv temporary tree variable
  <sa scanner attribute         <rn -- not used
  <st star op                   <nn typecast
  <tk token defn, call          <nc nonterminal call
  <as assignment                <co type constant }

funct {predefined in virtual compiler machine}
newsetlist ^int; {allocates set list, returns empty set}
newnewset !int ^int; {creates empty set in set list}
uniquenew !int ^int; {return first =set, dispose old if dupe}
addnewset !int !int; {add value to set in place}
newinset !int !int;

settree !set ^tree;
treeset !tree ^set;
treeint !tree ^int; {returns exp from <exp>}
inttree !int ^tree;
treetab !tree ^table;
treecopy !tree ^tree; {to get past tag compiler type fault}
charval ^int;
```

① 附录 B 的代码清单存在一些缺陷，有兴趣的读者可参阅作者 T. Pittman 新开发的 TAG 编译程序（该编译程序改为生成 C 语言而不是 Modula-2 语言的代码），详见：<http://www.ittybittycomputers.com/IttyBitty/TAGC/TAGinfo.html>。


```

[trans@=<tk otran xstt <no>aname >entry])+ ";" )+)?

("parser" (parsrule !data ^data@)+)?
("transformer" (tagrule !data ^data@)+)?
buildscanner !data ^pdata ^nst ^ncs
buildparser !pdata ^pdata
[bdata:<no benv bdef bred btok btran>]
[/"There is no goal symbol"/; from !tagname !benv ^<nt ..>]
finishoutput !tagname !nst !ncs
"end" ID ^endname [endname=tagname] ". ";

decln !insym:table !indef:set ^outsym:table ^outdef:set
-> "type" [outsym=insym; outdef=indef]
  (ID ^aname
    [/"type name already defined"/]
    [notinset !aname !outdef; addset !aname !outdef ^outdef@]
    [into !aname !outsym !tipe:<ty>aname ^outsym@]
    (" (ID^idn
      [/"value already defined"/]
      [notinset !idn !outdef; addset !idn !outdef ^outdef@]
      [into !idn !outsym !<co tipe >idn ^outsym@])$"," " )? )$"," " ";

-> "global" [outsym=insym; outdef=indef]
  (ID ^consname
    [/"global name already defined"/]
    [notinset !consname !outdef; addset !consname !outdef ^outdef@]
    (":" typename !outsym ^ctype [value=consname]
    ["=" (NUM^value [idn=1] | "<" [idn=3; value=<>]
      ["false" [idn=2; value=false] | "true" [idn=2; value=true])
    [from !idn !outsym@ ^ctype])
    [into !consname !outsym !<gl ctype value> ^outsym@]
    globalname !consname !ctype ";")+

-> "funct" [outsym=insym; outdef=indef]
  (ID ^functname [inh=<>; der=<>]
    [/"function name already defined"/]
    [notinset !functname !outdef; addset !functname !outdef ^outdef@]
    ("!" typename !outsym ^ity
      [inh@=<at inh <> ity >]) *
    ("^" typename !outsym ^dty
      [der@=<at der <> dty >]) *
    [into !functname !outsym !<fn inh der> ^outsym@] ";")+);

typename !env:table ^atype:tree
-> ID ^idn [/"invalid or undefined type"/; from !idn !env ^atype:<ty>]
-> "table" [from !5 !env ^atype]
-> "set" [from !4 !env ^atype]
-> "int" [from !1 !env ^atype]
-> "bool" [from !2 !env ^atype]
-> "tree" [from !3 !env ^atype];

scanre !indata:table !fromst:int !toknno:int ^outdata:table ^tost:int
-> scanalt !indata !fromst !toknno ^outdata ^tost
  ("|" splitalt !indata !outdata ^xdata
    scanalt !xdata !fromst !toknno ^tdata ^xst
    newtran !tran !xst !tost !empty !<> !toknno ^outtran
    joinalt !outdata !tdata ^outdata@)*;

scanalt !indata:table !fromst:int !toknno:int ^outdata:table ^tost:int
-> [tost=fromst; outdata=indata]
  (scanterm !outdata !tost !toknno ^outdata@ ^tost@)*;

scanterm !indata:table !orgst:int !toknno:int ^outdata:table ^tost:int
-> ("( unpackatts !indata ^inenv ^indef ^inred ^intok ^intran ^inhist
  [fromst=inhist+1]
  newtran !intran !orgst !fromst !empty !<> !toknno ^itran
  initer !data ^xdata
  scanre !xdata !fromst !toknno ^odata ^xst ")")

```

```

([outhist=ohist+1; tost=outhist]
newtran !xtran !xst !tost !empty !<> !toknno ^otran
(" " newtran !otran !xst !fromst !empty !<> !toknno ^ntran
  newtran !ntran !fromst !xst !empty !<> !toknno ^outran [kind=3]
|" " newtran !otran !xst !fromst !empty !<> !toknno ^outran [kind=2]
|"? " newtran !otran !fromst !xst !empty !<> !toknno ^outran [kind=1])
exiter !kind !indata !odata ^outdata
!exiter !0 !indata !odata ^outdata
  [tost=xst])
-> CHR^loch
unpackatts !indata ^env ^defd ^redef ^tok ^tran ^hist
  [addset !loch !empty ^onch; tost=hist+1]
(" " CHR^high ([loch<high; loch%=loch+1; addset !loch% !onch ^onch%])*)?
newtran !tran !orgst !tost !onch !<> !toknno ^ntran
packattributes !env !defd !redef !tok !ntran !tost ^outdata
-> [tdata=indata; acts=<>]
[" (scanact !tdata !toknno ^tdata% ^anact [acts%=<ca acts anact >])$";
"]" unpackatts !tdata ^env ^defd ^xred ^xtok ^xtran ^hist [tost=hist+1]
newtran !xtran !orgst !tost !empty !acts !toknno ^outran
packattributes !env !defd !xred !xtok !outran !tost ^outdata;

CHR ^char:int
-> STR ^strg
  [length !strg ^strlen]
  ([strlen=0; char=0] [strlen=1; charfrom !1 !strg ^char]);

scanact !indata:table !toknno:int ^outdata:table ^anact:tree
-> ID^name unpackatts !indata !inenv !indef ^redef ^tok ^tran ^hist
  ([["@ " [notinset !name !redef; inset !name !indef; tenv=inenv]
    [/"invalid iterator reference"/]
    [[from !name !tenv ^thev:<at ..>]
    | [from !name !tenv ^thev:<sa ..>]
    | [from !name !tenv ^thev:<vr ..>]]
    [ndef=indef; addset !name !redef ^nred; vtran=tran]
    | [[nothere !name !inenv; into !name !inenv !thev:<sa toknno>^name ^tenv]
    | [tenv=inenv; from !name !tenv ^thev:<at ..>; vtran=tran]]
    [/"name already defined"/]
    [notinset !name !indef; addset !name !indef ^ndef; nred=redef])
  packattributes !tenv !ndef !nred !tok !vtran !hist ^odata
  "= " expn !0 !odata ^outdata ^exptr ^expty
  [anact=<cas exptr expty thev>]
  | [/"invalid action call"/]; from !name !inenv ^fun:<fn inh der>]
  sends !inh !indata !<> ^sarg ^sdata
  recvs !der !sdata !<> !toknno ^rarg ^outdata
  [anact=<fn sarg rarg >^name]);

parsrule !indata:table ^outdata:table
-> [outdata=indata]
  ID^name [der=<>; inh=<>; ldata=outdata]
  (inherits !inh !ldata ^inh% ^ldata%)*
  (derives !der !ldata !0-1 ^der% ^ldata%)*
  "->" parsre !ldata ^rdata ^ptree
  (">" splitalt !ldata !rdata ^xdata
  parsre !xdata ^xdata ^prp
  joinalt !rdata !sdata ^rdata%
  [ptree=<al ptree prp >])*
  [ttree=<nt <no inh der >%0 ptree >^name]
  "; " datamerge !outdata !rdata ^xdata
  newdefine !name !xdata !ttree ^outdata% ;

parsre !indata:table ^outdata:table ^outree:tree
-> parsalt !indata ^outdata ^outree
  ("|" splitalt !indata !outdata ^xdata
  parsalt !xdata ^xdata ^nutree
  joinalt !outdata !xdata ^outdata%
  [outree=<al outree nutree >])*

```



```

parsalt !indata:table ^outdata:table ^outree:tree
-> [outdata=indata; outree=<>]
(parsterm !outdata ^outdata@ ^nutree
[outree@=<ca outree nutree >]]*);

parsterm !indata:table ^outdata:table ^outree:tree
-> initer !indata ^ndata
parsfact !ndata ^odata ^xtree
["*" [kind=3; outree=<st xtree >; xdata=odata]
["+ " [kind=2; outree=<pl xtree >; xdata=odata]
["?" [kind=1; outree=<al xtree <>; xdata=odata]
["$ " partoken !odata ^xdata ^nutree ^tokn
[kind=4; outree=<dl xtree nutree >%tokn ]
|[kind=0; outree=xtree; xdata=odata]]
exiter !kind !indata !xdata ^outdata
-> ID^tname
([from !tname !indata ^<tk der >%tokn]
recvs !der !indata !<> !0-1 ^rarg ^outdata
[outree=<tk rarg >%tokn]
| semantix !tname !<> !indata ^outdata ^outree )
-> semantix !0 !<> !indata ^outdata ^outree ;

parsfact !indata:table ^outdata:table ^outree:tree
-> "(" parsre !indata ^outdata ^outree ")"
-> partoken !indata ^outdata ^outree ^tokn;

partoken !indata:table ^outdata:table ^outree:tree ^newtok:int
-> STR^strng
unpackatts !indata ^env ^indef ^redef ^intok ^otran ^hist
[/["empty token string"//; length !strng ^len; len>0]
[newtok=strng; outree=<tk <>>%newtok]
([notinset !strng !intok; addset !newtok !intok ^outok]
[charno=0; xst=0]
([charno<len; charfrom !charno !strng ^thisch; charno@=charno+1]
[addset !thisch !empty ^onch; hist@=hist+1]
newtran !otran !xst !hist@ !onch !<> !0 ^otran@ [xst@=hist@])
[entry=<tk <>>%newtok]
[ittree !xst ^xxst; outran=<tk otran xxst <st>%strng >%entry]
|[otherwise; outok=intok; outran=otran])
packattributes !inenv !indef !redef !outok !outran !hist ^outdata;

tagrule !indata:table ^outdata:table
-> [outdata=indata]
ID^name [der=<>; inh=<>; ldata=outdata]
(inherits !inh !ldata ^inh@ ^ldata@)*
(derives !der !ldata !0-1 ^der@ ^ldata@)*
"->" tagre !ldata ^rdata ^ptree
(">" xformto !ptree !rdata ^rdata@ ^ptree@ )?
("->" splitalt !ldata !rdata ^xdata
tagre !xdata ^zdata ^prp
(">" xformto !prp !zdata ^zdata@ ^prp@ )?
joinalt !rdata !zdata ^rdata@
[ptree@=<al ptree prp >]]*
[ttree=<nt <no inh der >%l ptree >%name]
";" datamerge !outdata !rdata ^exdata
newdefine !name !exdata !ttree ^outdata@ ;

tagre !indata:table ^outdata:table ^outree:tree
-> tagalt !indata ^outdata ^outree
(["|" splitalt !indata !outdata ^xdata
tagalt !xdata ^zdata ^nutree
joinalt !outdata !zdata ^outdata@
[outree@=<al outree nutree >]]*);

tagalt !indata:table ^outdata:table ^outree:tree
-> "(" tagre !indata ^outdata ^outree ")"
-> recogtree !0 !1 !<> !indata !0-1 ^outdata ^comp

```

```

[outree=<mt comp>]
(semantix !0 !outree !outdata ^outdata@ ^outree@ );

xformto !intree:tree !indata:table ^outdata:table ^outree:tree
-> expn !0 !indata ^outdata ^etree ^exptype
lookup !3 !indata ^exptype
[outree=<ca intree <xf etree>>]
(semantix !0 !outree !outdata ^outdata@ ^outree@ );

semantix !idn:int !intree:tree !indata:table ^outdata:table ^outree:tree
-> [idn#0; indata:<no inenv indef inred intok intran >%inhist]
lookup !3 !indata ^trtr
( ":" ID^ntname
  ( [from !idn !indata ^tren:<vr trty>]
    | [from !idn !indata ^tren:<at lnk vrv trty>])
    ( [trty=trtr] | [trty=<>] tren:retype !trtr )
    | [ntname=idn; tren=<>] )
  ( [from !ntname !indata ^ntdef:<nt <no inh der >%kind ptree >]
    ([ntname=idn; kind=0] | [ntname#idn; kind=1])
    | [inh=<no >; der=inh] )
  sends !inh !indata !<> ^sarg ^tdata
  recvs !der !tdata !<> !0-1 ^rarg ^outdata
  [outree=<ca intree ntree:<nt sarg rarg tren >%ntname >]
-> "["
  [outree=intree]
  (action !outdata !0-1 ^outdata@ ^nutree
    [outree=<ca outree nutree >])$";" "]"
-> ID^ntname [idn=0]
  semantix !tname !intree !indata ^outdata ^outree ;

initer !indata:table ^outdata:table
-> unpackatts !indata ^env ^defd ^redef ^tok ^tran ^hist
  packattributes !env !defd !empty !tok !tran !hist ^outdata;

exiter !kind:int !indata:table !exdata:table ^outdata:table
-> unpackatts !indata ^inenv ^indef ^redef ^tok ^tran ^hist
  unpackatts !exdata ^xenv ^xdef ^xred ^xtok ^xtran ^xhist
  ([[kind=kind/2*2; odef=xdef]][kind>kind/2*2; union !indef !xred ^odef])
  ([[kind>0]][kind=0; //"@-identifiers not in iterator"/]; xred=empty])
  [ored=redef] {*** used to be: union !redef !xred ^ored ***}
  [difference !xdef !indef ^nudef; intersect !nudef !xred ^empty]
  packattributes !xenv !odef !ored !xtok !xtran !xhist ^outdata;

splitalt !indata:table !exdata:table ^outdata:table
-> unpackatts !indata ^inenv ^indef ^redef ^tok ^tran ^hist
  unpackatts !exdata ^xenv ^xdef ^xred ^xtok ^xtran ^xhist
  packattributes !inenv !indef !redef !xtok !xtran !xhist ^outdata;

joinalt !indata:table !exdata:table ^outdata:table
-> unpackatts !indata ^inenv ^indef ^redef ^tok ^tran ^hist
  unpackatts !exdata ^xenv ^xdef ^xred ^xtok ^xtran ^xhist
  [intersect !xdef !indef ^odef; intersect !xred !redef ^ored]
  packattributes !xenv !odef !ored !xtok !xtran !xhist ^outdata;

newdefine !name:int !indata:table !value:tree ^outdata:table
-> unpackatts !indata ^inenv ^indef ^redef ^tok ^tran ^hist
  [/"name already defined"/]
  [nothere !name !inenv; notinset !name !indef]
  [into !name !inenv !value ^outenv; addset !name !indef ^outdef]
  [outran=<nt intran value >%name]
  packattributes !outenv !outdef !redef !tok !tran !hist ^outdata;

derives !inatt:tree !indata:table !toknno ^outatt:tree ^outdata:table
-> "A" unpackatts !indata ^inenv ^def ^redef ^tok ^tran ^hist
  ID^idn ":" typename !inenv ^tipe
  [/"attribute name already defined"/]
  [nothere !idn !inenv; notinset !idn !def]

```

```

    [outatt=<at inatt <tk>%toknno tipe>%idn;
    into !idn !inenv !outatt ^outenv]
    packattributes !outenv !def !redef !tok !tran !hist ^outdata;

inherits !inatt:tree !indata:table ^outatt:tree ^outdata:table
-> (!" unpackatts !indata ^inenv ^indef ^redef ^tok ^tran ^hist
ID^idn ":" typename !inenv ^tipe
[notinset !idn !indef; addset !idn !indef ^outdef]
[/"attribute name already defined"/; nothere !idn !inenv]
[outatt=<at inatt <> tipe>%idn; into !idn !inenv !outatt ^outenv]
packattributes !outenv !outdef !redef !tok !tran !hist ^outdata;

sends !formal:tree !indata:table !intree:tree ^actual:tree ^outdata:table
-> ( [formal:<no>]
    (!" expn !0 !indata ^tdata ^exptr ^expty
    sends !formal !tdata !<at intree exptr expty> ^actual ^outdata
    |[otherwise; outdata=indata; actual=intree] )
    | [formal:<at nextf vrv tipe >]
    sends !nextf !indata !intree ^xtree ^xdata
    !" expn !0 !xdata ^outdata ^exptr ^expty
    ([expty#<>; /"attribute type mismatch"/; expty=tipe])?
    [actual=<at xtree exptr tipe>]
    | [formal=<>; outdata=indata; actual=intree]);

recvs !formal:tree !indata:table !intree:tree !toknno:int
^actual:tree ^outdata:table
-> ( [formal:<no>; tipe=<>] lookup !3 !indata ^trty
    ( "^^
    (ID ^idn
    (":"
    newvariable !true !idn !indata !trty !toknno ^tdata ^thevar ^wasdef ^vartype
    recogtree !0 !1 !thevar !tdata !toknno ^odata ^comp
    [exptr=<ca thevar <mt comp >>; wasdef=<>]
    |newvariable !true !idn !indata !tipe !toknno ^tdata ^thevar ^wasdef ^vartype
    ([wasdef=<>; odata=tdata; exptr=thevar]
    |[otherwise; odata=tdata;
    exptr=<ca thevar <mt <eq wasdef thevar >>>])
    |newvariable !true !0-1 !indata !trty !toknno ^tdata ^thevar ^wasdef ^vartype
    recogtree !0 !1 !thevar !tdata !toknno ^odata ^comp
    [exptr=<ca thevar <mt comp >>])
    recvs !formal !odata !<at intree exptr vartype> ^toknno ^actual ^outdata
    |[otherwise; outdata=indata; actual=intree])
    | [formal:<at nextf vrv tipe >; trty=tipe]
    recvs !nextf !indata !intree !toknno ^xtree ^tdata "^^
    ( ID ^idn
    (":"
    newvariable !true !idn !tdata !trty !toknno ^odata ^thevar ^wasdef ^vartype
    recogtree !0 !1 !thevar !odata !toknno ^outdata ^comp
    [exptr=<ca thevar <mt comp >>; wasdef=<>]
    |newvariable !true !idn !tdata !tipe !toknno ^odata ^thevar ^wasdef ^vartype
    ([wasdef=<>; outdata=odata; exptr=thevar]
    |[otherwise; outdata=odata;
    exptr=<ca thevar <mt <eq wasdef thevar >>>])
    | newvariable !true !0-1 !tdata !trty !toknno ^odata ^thevar ^wasdef ^vartype
    recogtree !0 !1 !thevar !odata !toknno ^outdata ^comp
    [exptr=<ca thevar <mt comp >>])
    [actual=<at xtree exptr vartype>]
    | [formal=<>; outdata=indata; actual=intree]);

action !indata:table !toknno:int ^outdata:table ^outree:tree
-> ID^name (assigns or asserts or predeclared function call)
[indata:<no inenv indef redef tok tran >%hist]
( "0" ([notinset !name !redef; asn=true])|[otherwise; asn=false])
[inset !name !indef] [/"invalid iterator reference"/]
[([from !name !inenv ^thev:<at ..>])|[from !name !inenv ^thev:<vr ..>]
|[from !name !inenv ^thev:<sa ..>])
|[notinset !name !indef; asn=true]

```

```

    ([nothere !name !inenv] [[from !name !inenv ^<at ...>]]
    |[otherwise; asn=false]) )
  ( [asn=true]
    "=" expn !0 !indata ^tdata ^exptr ^expty
    newvariable !true !name !tdata !expty !toknno ^outdata ^thev ^ovar ^vartype
    ([ovar=<>; outtree=<as exptr expty thev >]
    |[otherwise; outtree=<as exptr expty ovar >])
  | [from !name !inenv ^fun:<fn inh der>; //"invalid action call"//]
    sends !inh !indata !<> ^sarg ^sdata
    recvs !der !sdata !<> !toknno ^rarg ^outdata
    [outtree=<fn sarg rarg >name]
  | [otherwise; from !name !inenv ^fun; //"invalid assertion"//]
    ([fun:<at ...>] [[fun:<vr ...>] [[fun:<gl ...>] [fun:<sa ...>]])
    (":" recogtree !0 !1 !fun !indata !toknno ^outdata ^btree
    |boolex !name !indata ^outdata ^btree)
    [outtree=<mt btree >] )
-> STR^strg                                     (output text)
    [outdata=indata; outtree=<ot >%strg]
-> boolex !0 !indata ^outdata ^btree             (any assertion)
    [outtree=<mt btree >]
-> //" [outdata=indata; etree=<>]                 (error action for failure)
    (erract !outdata !toknno ^outdata^ ^btree
    [etree=<ca etree btree >])$";
    //" [outtree=<er etree >];

erract !indata:table !toknno:int ^outdata:table ^outtree:tree
-> STR^strg                                     (output text)
    [outdata=indata; outtree=<ot >%strg]
-> ID^name                                       (predeclared function call or assigns only)
unpackatts !indata ^inenv ^indef ^redef ^tok ^tran ^hist
( ("@" [notinset !name !redef; inset !name !indef; tenv=inenv]
  //"invalid iterator reference"//)
  ([from !name !tenv ^thev:<at ...>] [[from !name !tenv ^thev:<vr ...>]
  |[from !name !tenv ^thev:<sa ...>])
  [ndef=indef; addset !name !redef ^nred]
  |([nothere !name !inenv]
    ([toknno=0-1; into !name !inenv !thev:<vr <>>%name ^tenv]
    |[otherwise; into !name !inenv !thev:<sa toknno <>>%name ^tenv])
    |[tenv=inenv; from !name !tenv ^thev:<at ...>])
    //"name already defined"//)
    [notinset !name !indef; addset !name !indef ^ndef; nred=redef])
  packattributes !tenv !ndef !nred !tok !tran !hist ^sdata
  "=" expn !0 !sdata ^outdata ^exptr ^expty
  [outtree=<as exptr expty thev >]
  ([from !name !inenv ^iden] ([iden:<vr <>>] [[iden:<sa xtk <>>])
  iden:reddecorate !name !expty?      {I hope this is the same one!})
  | [/"invalid action call"//; from !name !inenv ^fun:<fn inh der>]
    sends !inh !indata !<> ^sarg ^sdata
    recvs !der !sdata !<> !toknno ^rarg ^outdata
    [outtree=<fn sarg rarg >name]);

recogtree !path:int !depth:int !root:tree !indata:table !toknno:int
  ^outdata:table ^exptree:tree {*** limit depth to 32 bits ***}
-> "..." lookup !2 !indata ^exptype (don't care; return true)
    [outdata=indata; exptree=<cn exptype >%1]
-> ID ^idn lookup !3 !indata ^tipe
  newvariable !true !idn !indata !tipe !toknno ^tdata ^thear ^wasdef ^vartype
  [inttree !path ^pathtree; inttree !0 ^compare; wasdef=<>]
  (":" recogtree !path !depth !root !tdata !toknno ^outdata ^comp
  [exptree=<an <rt root pathtree compare thevar > comp >]
  | [exptree=<rt root pathtree compare thevar >; outdata=tdata])
-> "<>" (empty tree)
    [inttree !path+9^depth ^pathtree; inttree !-1 ^compare] {9 sb 10??}
    [outdata=indata; exptree=<rt root pathtree compare <>>]
-> "<" ID ^idn (might could take variable node name? maybe later)
  [/"invalid node name"//]
  lookup !idn !indata ^<co ...>

```

```

[inttree !path+9*depth ^pathtree; inttree !idn ^compare] {9 sb 10??}
[subtr=<rt root pathtree compare <>>; edata=indata; index=1]
(recogtree !path+index*depth !depth*10 !root !edata !toknno ^edata@ ^etree
 [subtr@=<an subtr etree >; index@=index+1])* ">"
("%" ID ^idn lookup !3 !indata ^ttype
 newvariable !false !idn !edata !ttype !toknno ^edata@ ^thevar ^wasdef ^vartype
 [inttree !path+8*depth ^decorpath; wasdef=<>]
 [subtr@=<an subtr <rt root decorpath compare thevar >>]]?
 [exptree=subtr; outdata=edata] ;

boolex !iname:int !indata:table ^outdata:table ^exptree:tree
-> "otherwise"
  [/"syntax error"/; iname=0; lookup !2 !indata ^exptype]
  [outdata=indata; exptree=<cn exptype >#1]
-> expn !iname !indata ^outdata ^exptree ^exptype
  lookup !2 !outdata ^exptype;

expn !iname:int !indata:table ^outdata:table ^exptree:tree ^exptype:tree
-> boolterm !iname !indata ^outdata ^exptree ^xtype
  (("" boolterm !0 !outdata ^outdata@ ^btree ^otype (or)
   checktype !2 !otype !indata lookup !2 !indata ^exptype
   [exptree=<or exptree btree >]) +
  |[exptype=xtype]);

boolterm !iname:int !indata:table ^outdata:table ^exptree:tree ^exptype:tree
-> boolfact !iname !indata ^outdata ^exptree ^xtype
  (("s" boolfact !0 !outdata ^outdata@ ^btree ^atype (and)
   checktype !2 !atype !indata lookup !2 !indata ^exptype
   [exptree=<an exptree btree >]) +
  |[exptype=xtype]);

boolfact !iname:int !indata:table ^outdata:table ^exptree:tree ^exptype:tree
-> "~" [iname=0] boolfact !0 !indata ^outdata ^btree ^exptype (not)
  checktype !2 !exptype !outdata
  [exptree=<no btree >]
-> sexpn !iname !indata ^edata ^etree ^etype
  ("=" sexpn !0 !edata ^outdata ^btree ^etype
   lookup !2 !outdata ^exptype
   [exptree=<eq etree btree >]
  "<" sexpn !0 !edata ^outdata ^btree ^etype
   lookup !2 !outdata ^exptype
   [exptree=<ls etree btree >]
  ">" sexpn !0 !edata ^outdata ^btree ^etype
   lookup !2 !outdata ^exptype
   [exptree=<gr etree btree >]
  "#" sexpn !0 !edata ^outdata ^btree ^etype
   lookup !2 !outdata ^exptype
   [exptree=<ne etree btree >]
  |[exptree=etree; exptype=etype; outdata=edata]);

sexpn !iname:int !indata:table ^outdata:table ^exptree:tree ^exptype:tree
-> ( [iname=0] "+" term !0 !indata ^outdata ^exptree ^exptype
  checktype !1 !exptype !outdata lookup !1 !indata ^xtype
  |[iname=0] "-" term !0 !indata ^outdata ^etree ^exptype
  checktype !1 !exptype !outdata lookup !1 !indata ^xtype
  [exptree=<ng etree >]
  | term !iname !indata ^outdata ^exptree ^xtype)
  ( ("+" term !0 !outdata ^outdata@ ^etree ^pltype
   checktype !1 !pltype !outdata lookup !1 !indata ^exptype
   [exptree=<ad exptree etree >]
  "- term !0 !outdata ^outdata@ ^etree ^mtype
   checktype !1 !mtype !outdata lookup !1 !indata ^exptype
   [exptree=<su exptree etree >]) +
  |[exptype=xtype]);

term !iname:int !indata:table ^outdata:table ^exptree:tree ^exptype:tree
-> fact !iname !indata ^outdata ^exptree ^exptype

```

```

(("*" fact !0 !outdata ^outdata@ ^etree ^xtype
  checktype !1 !xtype !outdata lookup !1 !indata ^exptype
  [exptree=<cmp exptree etree >]
  |"/" fact !0 !outdata ^outdata@ ^etree ^xtype
  checktype !1 !xtype !outdata lookup !1 !indata ^exptype
  [exptree=<dv exptree etree >]
  |\" fact !0 !outdata ^outdata@ ^etree ^xtype (mod)
  checktype !1 !xtype !outdata lookup !1 !indata ^exptype
  [exptree=<md exptree etree >]) +
  | [exptype=exptype]};

fact !iname:int !indata:table ^outdata:table ^exptree:tree ^exptype:tree
-> ":" [/"syntax error"/; iname#0] (named tree)
fact !0 !indata ^<no env defd redef tok tran >%hist ^etree ^exptype
  [/"tree name already defined"/]
  [notinset !iname !defd; addset !iname !defd ^outdef]
  ([nothere !iname !env; into !iname !env !thex:<vr exptype >%iname ^outenv]
  |[otherwise; outenv=env]
  [/"invalid node name"/; from !iname !env ^thex:<at lnk vrv exptype >])
  [/"invalid named tree expression"/]
  [from !3 !outenv ^exptype; from !iname !outenv ^ref]
  [outdata=<no outenv outdef redef tok tran >%hist]
  [exptree=<ca <as etree exptype thev > ref >]
-> [iname#0; outdata=indata]
  unpackatts !indata ^env ^defd ^redef ^tok ^tran ^hist
  [from !iname !indata ^exptree]
  [/"undefined identifier"/; inset !iname !defd]
  [/"invalid identifier in expression"/]
  ([exptree:<at lnk vrv exptype >] |[exptree:<vr exptype>]
  |[exptree:<gl exptype>] |[exptree:<sa xtk exptype>])
-> "<" [/"syntax error"/; iname=0] (empty tree)
  [outdata=indata; lookup !3 !indata ^exptype; exptree=<bt <> <>>]
-> "<" [/"syntax error"/; iname=0] lookup !3 !indata ^exptype
  [/"invalid node name"/]
  ID ^idn lookup !idn !indata ^<co ..>
  [subtrees=<bt <> <>>; edata=indata; index=1; decor=<bt <> <>>]
  (fact !0 !edata ^edata@ ^etree ^xxtype (** not LL(1) **))
  (** also, the xxtype makes repair so it compares it to exptype **)
  [subtrees=<ca subtrees etree >%index; index=index+1]) * ">"
  ("% sexpn !0 !edata ^edata@ ^dtree ^etype
  [decor=<nn dtree etype exptype >])?
  [exptree=<bt subtrees decor >%idn; outdata=edata]
-> "false"
  [/"syntax error"/; iname=0; lookup !2 !indata ^exptype]
  [outdata=indata; exptree=<cn exptype >%0]
-> "true"
  [/"syntax error"/; iname=0; lookup !2 !indata ^exptype]
  [outdata=indata; exptree=<cn exptype >%1]
-> "empty"
  [/"syntax error"/; iname=0; lookup !4 !indata ^exptype]
  [outdata=indata; exptree=<cn exptype >%0]
-> "vacant"
  [/"syntax error"/; iname=0; lookup !5 !indata ^exptype]
  [outdata=indata; exptree=<cn exptype >%0]
-> NUM ^value
  [/"syntax error"/; iname=0; lookup !1 !indata ^exptype]
  [outdata=indata; exptree=<cn exptype >%value]
-> ID ^name
  [/"syntax error"/; iname=0]
  [indata:<no env defd redef tok tran >%hist]
  ("% [/"invalid use of iterator"/; inset !name !redef]
  | [/"invalid use of iterator"/; notinset !name !redef])
  fact !name !indata ^outdata ^exptree ^exptype
-> "(" [iname=0] expn !0 !indata ^outdata ^exptree ^exptype " ");

newvariable !chktyp:bool !iname:int !indata:table !tipe:tree !toknno:int
  ^outdata:table ^thevar:tree ^oldvar:tree ^vartype:tree

```

```

-> [name<0]
  unpackatts !indata ^inenv ^indef ^redef ^tok ^tran ^hist
  packattributes !inenv !indef !redef !tok !tran !hist+1 ^outdata
    [thevar=<tv tipe >%idn; oldvar=<> vartype=tipe]
-> [name>0]
  unpackatts !indata ^inenv ^indef ^redef ^tok ^tran ^hist
  ("@" (** what about call from assignment? **)
    [/"invalid iterator reference"/; inset !name !indef]
    [[from !name !inenv ^avar:<at lnk xrv exty>]
    |[from !name !inenv ^avar:<vr exty>]
    |[from !name !inenv ^avar:<sa xtk exty>]]
    ([chktyp=true; tipe#<>; exty#<>;
    [/"attribute type mismatch"/; exty=tipe]]?)
    ([inset !name !redef; oldvar=avar]
    newvariable !chktyp !0-1 !indata !exty !toknno ^outdata ^thevar ^xvar ^vartype
    | [otherwise; addset !name !redef ^nred]
    packattributes !inenv !indef !nred !tok !tran !hist ^outdata
      [thevar=avar; oldvar=<>; vartype=tipe])
    |([inset !name !indef]
      [[from !name !inenv ^oldvar:<at lnk xrv exty>]
      |[from !name !inenv ^oldvar:<vr exty>]
      |[from !name !inenv ^oldvar:<sa xtk exty>]]
      ([chktyp=true; tipe#<>; exty#<>;
      [/"attribute type mismatch"/; exty=tipe]]?)
      newvariable !chktyp !0-1 !indata !exty !toknno ^outdata ^thevar ^xvar ^vartype
      | [otherwise; addset !name !indef ^nred; oldvar=<>]
      ([nothere !name !inenv; vartype=tipe]
      ([toknno=0-1; into !name !inenv !thevar:<vr tipe>%name ^tenv]
      |[otherwise; into !name !inenv !thevar:<sa toknno tipe>%name ^tenv])
      [exty=tipe] packattributes !tenv !indef !redef !tok !tran !hist ^outdata
      |[otherwise; tenv=inenv; from !name !tenv ^thevar:<at lnk vrv vartype>]
      packattributes !tenv !indef !redef !tok !tran !hist ^outdata
      ([chktyp=true; tipe#<>; [/"attribute type mismatch"/; vartype=tipe]]?))
-> [otherwise]
  ID^idn
  newvariable !chktyp !idn !indata !tipe !toknno ^outdata ^thevar ^oldvar ^vartype;

lookup !name:int !data:table ^value:tree
-> [data:<no env defd redef tok tran >%hist; from !name !env ^value];

checktype !mustbe:int !typ:tree !data:table
-> [/"wrong type"/] lookup !mustbe !data ^typ;

newtran !intran:tree !fromst:int !tost:int !onch:set
  !actn:tree !tokn:int ^outtran:tree
-> [inttree !fromst ^ftrtree; inttree !tost ^totree; settree !onch ^ontree]
  [outtran=<tr intran ftrtree totree ontree actn>%tokn];

datamerge !ldata:table !rdata:table ^outdata:table
-> unpackatts !ldata ^inenv ^indef ^redef ^tok ^tran ^hist
  unpackatts !rdata ^xenv ^xdef ^xred ^xtok ^xtran ^xhist
  packattributes !inenv !indef !ored !xtok !xtran !xhist ^outdata;

uniqueset !oldlist:tree !theset:set ^setno:int ^newlist:tree
[find or create a set item in the list, =theset, return index]
-> [alist=oldlist; newlist=oldlist; setno=0; settree !theset ^asett]
  ([asett#<>]
  ([alist:<no link tset >%ixt; treeset !tset ^aset]
  [alist@=link; difference !theset !aset ^dif1]
  [difference !aset !theset ^dif2]
  ([dif1=dif2; treeint !ixt ^ixn; setno=@ixn; alist@=<>]]? )*)
  ([setno=0]
  ([oldlist:<no ...>%ixt; treeint !ixt ^ixn][otherwise; ixn=0])
  [setno=@ixn+1; newlist@=<no oldlist asett >%ixn+1]]? );

nothertranxx !atran:tree !frosty:int !tosty:int !aset:set !actor:int ^trans:tree
-> [inttree !frosty ^ftrtree; inttree !tosty ^totree]
  [settree !aset ^seton; inttree !actor ^acton]

```

```

[trans=<tr atran ftree totree seton action >];

scannerform !thelist:tree !defaults:set !inchs:tree
^chrsets:tree ^outacts:tree ^actno:int ^matchtoks:tree
{thelist is a list of tkstr nodes built in scanner & newtran}
{defaults is the set of all chars, diminished by any particular }
{ chars along the way, to become "anychar" in strings & comments}
{inchs accumulates disjoint character subsets, which becomes }
{ chrsets on the way out, sets of which tag revised transitions}
{outacts is a list of unique action sequences, one each per tran}
{matchtoks is an isolated list of tk nodes, from which to build }
{ the matchtok procedures } {trans is a cleaned-up list of tr }
{ transitions, including converts from tk "on char 256"}
-> [thelist:<tr link ftree totree ontree acts >; elt=0; nelts=0]
[treaset !ontree ^onset; unset=onset; treaset !<> ^empty]
[[onset#empty]
  ([elt<256; anelt=elt; elt@=anelt+1]
    ([inset !anelt !onset; nelts@=nelts+1]
      ([notinset !anelt+1 !onset; elt@=260])? )? )? )?
    ([nelts<64; difference !defaults !onset ^redefs]
      |[otherwise; redefs=defaults])
    [alist=inchs; nxlist=inchs]
    ([onset#empty; alist:<no slink tset >%ixt; treaset !tset ^aset]
      [difference !onset !aset ^dif1; difference !aset !onset ^dif2]
      [intersect !onset !aset ^dif0; onset@=dif1; nlist=slink]
      ([dif2#empty&dif2#aset; nlist@=alist] alist:deleteitem
        uniqueset !nxlist !dif0 ^setn0 ^xlist
        uniqueset !xlist !dif2 ^setn2 ^nxlist@ )?
      [alist@=nlist])*)
  uniqueset !nxlist !onset ^setn1 ^inchex
  scannerform !link !redefs !inchex ^chrsets ^inacts ^nactn ^matchtoks
  [csets=chrsets]
  ([acts=<>; actno=nactn; outacts=inacts; actor=0]
    |[otherwise; actno=nactn+1; actor=actno]
      [outacts=<no inacts <no <> acts >%actno >%actno])
  ([nelts<64; bset=empty]|[otherwise; addset !0 !empty ^bset])
  ([csets:<no clink cset >%cnot; treeint !cnot ^cno]
    [treaset !cset ^oset; csets@=clink]
    [intersect !unset !cset ^qset]
    ([qset#empty; addset !cno !bset ^nset; bset@=nset])? )? )?
  [treeint !ftree ^frosty; treeint !totree ^tosty]
  nothertran !frosty !tosty !bset !actor

-> [thelist:<tk link ftree nmtree >%acts; treaset !<> ^empty]
scannerform !link !defaults !inchs ^chrsets ^inacts ^nactn ^machs
(([nmtree:<no ...>; acts:<tk ders >%tokno]
  [matchtoks=<tk machs ftree nmtree >%acts]
  |[nmtree:<st>%tokno; ders=<>; matchtoks=machs])
  [addset !3 !empty ^cset; actno=nactn+1]
  [outacts=<no inacts <no <> <tr ders >%tokno >%actno >%actno]
  [treeint !ftree ^frosty; treeint !tokno ^tokn]
  nothertran !frosty !0 !cset !actno
  |[otherwise; matchtoks=machs; actno=nactn; outacts=inacts])

-> [thelist:<nt link ontree >]
  scannerform !link !defaults !inchs ^chrsets ^outacts ^actno ^matchtoks
-> [thelist:<nc link ontree >]
  scannerform !link !defaults !inchs ^chrsets ^outacts ^actno ^matchtoks
-> [thelist=<>; outacts=<>; actno=0; matchtoks=<>]
  [treaset !<> ^empty; addset !0 !empty ^rets; ones=inchs]
  [addset !32 !empty ^spas; settree !rets ^rett]
  [addset !256 !empty ^mgss; settree !mgss ^mgst]
  [settree !spas ^spat; settree !defaults ^deset]
  [csets=<no <no <no <no <> deset >%0 rett >%1 spat >%2 mgst >%3]
  ([ones:<no olink osett >; ones@=olink; treaset !osett ^oset]
    uniqueset !csets !oset ^setx ^csets@ )? )?
  [chrsets=csets];

```



```

findcycle !trans:int !headst:int !here:int !looky:set
{if you find an empty to headst, convert all statenos of trans }
{ in looky to headst; if from=to then delete; err if acts#<>}
-> [flist=trans; addset !here !looky ^looking; treeset !<> ^mtset]
(getatran !flist ^frono ^tono ^chrs ^acts ^flist@
  ([frono=here&chrs=mtset]          {flist,frono,tono,looking})
  ([tono=headst; alist=-1]
    (getatran !alist ^afro ^ato ^achrs ^acta ^alink
      [rev=false]                {##alist,afro,ato,achrs}
      ([inset !afro !looking; afro@=headst; rev@=true])?
      ([inset !ato !looking; ato@=headst; rev@=true])?
      ([ato=afro&achrs=mtset]    fixatran !alist !-1 !-1 !mtset !-1
        [/"Empty cycle action"/; acta=<>]
        |[rev=true] fixatran !alist !afro !ato !achrs !acta )
      [alist=alink]))*
  |[notinset !tono !looking]
  findcycle !-1 !headst !tono !looking ))? ) * ;

matchlist !atree:tree !btree:tree {fails if different}
-> [atree=btree]
-> [atree:<no alnk acode >%adec; treeint !adec ^anum]
[btree:<no blnk bcode >%bdec; treeint !bdec ^bnum]
[anum=bnum] matchlist !alnk !blnk;

cataction !oldact:tree !act1:int !act2:int ^act3:int ^newact:tree
{find or create action list act1;act2 (no dupes), return id# tree}
-> [alist=oldact; blist=oldact; newact=oldact; sub1=<>; sub2=<>]
[got=false; hino=0]
([act1=0; act3=act2]
|[act2=0; act3=act1]
|([alist:<no alnk aseq >%axt; treeint !axt ^axn]
  ([act1=axn; sub1@=aseq])? ([act2=axn; sub2@=aseq])?
  ([hino=axn; hino@=axn])? [alist=@alnk]))*
sub1:catlist !sub2 ^sub3
([blist:<no blnk bseq >%bxt; blist@=blnk; treeint !bxt ^bxtn]
  (matchlist !sub3 !bseq [act3=bxtn; blist@=<>; got@=true]))? ) *
([got=false; act3=hino+1]
  [newact@=<no oldact sub3 >%act3])? ) ;

buildscanner !indata:table ^outdata:table ^highstate:int ^hicharset:int
-> unpackatts !indata ^envt ^indef ^inred ^intok ^intran ^inhist
  ["CONST MuchTooBig=262143;TableHigh=32767;"]
  ["TYPE BigRange=[0..MuchTooBig];TableRange=[0..TableHigh];"]
  ["StateTableAry=ARRAY BigRange OF TableRange;"]
  ["StateTable=POINTER TO StateTableAry;"]
  ["VAR TokenPreview,CurrentState,ncharsets:INTEGER;"]
  ["scannextch:CHAR;StateTablePtr:StateTable;"]
  ["chartrans:ARRAY CHAR OF INTEGER;"]
  [treetab !envt ^env; treeset !<> ^mtset; allset=mtset; tch=1]
  ([tch<256; nch=tch; tch@=tch+1]
    ([nch#32&(nch<8^nch>13); addset !nch !allset ^allset@]))? ) *
scannerform !intran !allset !<>
  ^trans ^chrsets ^actsets ^actnx ^matchtoks
{e} [atran=trans; change=true] {actsets...}
actsets:findvars !env !mtset ^xset
["errno:INTEGER;"]
["PROCEDURE NewStateTable(nitems:BigRange);"] {generic}
["VAR ix:BigRange;BEGIN NEW(StateTablePtr);"]
["FOR ix:=0 TO MuchTooBig DO StateTablePtr^[ix]:=0 END END;"]
["PROCEDURE doerr;VAR ok:BOOLEAN;"]
["BEGIN ok:=TRUE;CASE errno OF 0:|"]
actsets:finderrs !env !1 ^lasterr
["END;IF errno>0 THEN abortit ELSE errno:=0;END;(*doerr*)"]
{remove empty cycles}
(getatran !atran ^cyc0 ^cycl ^chrs1 ^acts1 ^link1
  ([chrs1=mtset] findcycle !link1 !cyc0 !cycl !mtset ))?

```

```

    [atran@=link1])*
(remove empty moves, transfer action to successor moves)
{~)
  ([change=true; change@=false; btran=trans]
  (getatran !btran ^from3n ^froto ^chrs3 ^acts3 ^link3
  ([chrs3=mtset; ctran=trans; change@=true; chng=false]
  (getatran !ctran ^tofro ^tost4n ^chrs4s ^acts4 ^ctran@
  ([froto=tofro]
  cataction !actsets !acts3 !acts4 ^acts34 ^actsets@
  ([chng=false]
  fixatran !btran !from3n !tost4n !chrs4s !acts34
  |[otherwise]
  nothertran !from3n !tost4n !chrs4s !acts34 )
  [chng@=true]))? )? )?
  [btran@=link3])* )?
(convert to dfsm (no reduce))
{ }
  [chrsets:<no ..>%hinot; treeint !hinot ^hino; nxchs=chrsets]
  [newsetlist ^stsets; newnewset !stsets ^firstset]
  [addnewset !0 !firstset; doing=firstset]
  [more=true; newsetlist ^usacts]
  ([more=true; more@=false; alln=0] (:doing,alln)
  ([alln<hino+1; oldtr=trans; newnewset !stsets ^alls]
  [newnewset !usacts ^user]
  (getatran !oldtr ^frstn ^dest ^chrs8s ^actor ^oldtr@
  ([inset !alln !chrs8s; newinset !frstn !doing]
  ([actor#0; addnewset !actor !user]))?
  [addnewset !dest !alls]))? )?
  [uniquenew !alls ^destn; uniquenew !user ^actno]
  ([destn>firstset; dests=(destn-firstset)/4]
  |[otherwise; dests=0])
  ([dests>0 ^actno>3; "(*"; number!(doing-firstset)/4]
  [", "; number!dests; ", "; number!alln; ", "]
  [number!actno/4; "*)"; ascii!0]
  notherdetran !(doing-firstset)/4 !(destn-firstset)/4 !alln !actno )?
  [alln@=alln+1])*
{n)
  ([alls>doing+4 ^destn>doing; more@=true; doing@=doing+4]))? )?
(transmoglify halt/block transitions)
(^)
  [etran=-1]
  (getadetrans !etran ^frost ^tost5 ^ch5 ^acts5 ^link5
  ([ch5=3; ftran=-1; gtran=-1; chg=false; nset=mtset]
  (getadetrans !ftan ^from6 ^tosey ^ch6 ^acts6 ^ftan@
  ([frost=from6&tosey#tost5; addset !ch6 !nset ^nset; nset@=mnset]))? )?
  (getadetrans !gtran ^froy ^tost7n ^ch7 ^acts7 ^gtran@
  ([froy=tost5; notinset !ch7 !nset]
  [newunion !acts5 !acts7 ^acts57]
  ([chg=false; chg@=true]
  fixadetrans !etran !frost !tost7n !ch7 !acts57
  |notherdetran !frost !tost7n !ch7 !acts57 ))? )? )?
  [etran@=link5])*
(output dfsm)
{%)
  ["PROCEDURE scanstateitem (st,ch,ac,ns:INTEGER);"]
  ["VAR ix:BigRange;BEGIN ix:=st;ix:=(ix*ncharsets+ch)*2;"]
  ["StateTablePtr^[ix]:=ac;StateTablePtr^[ix+1]:=ns END scanstateitem;"]
  ["PROCEDURE ScannerFill;VAR ix:CHAR;BEGIN"]
  [htran=-1]
  (getadetrans !htran ^frost8 ^tost8 ^ch8 ^acts8 ^htran
  ([tost8>0 ^acts8>3; "scanstateitem(("; number!frost8; "),"]
  [number!ch8; ", "; number!acts8/4; ", "; number!tost8; ");"]
  [ascii!0]))? )? )?
(output rest of scanner code)
{")
  ["FOR ix:=CHR(0)TO CHR(255)DO chartrans[ix]:=0 END;"]
  ["FOR ix:=CHR(8)TO CHR(12)DO chartrans[ix]:=4 END;"] { :=6; }
  ["chartrans[CHR(13)]:=2;"]
  ([nxchs:<no nxlnk achsett >%setnt; treeint !setnt ^setno]
  [nxchs@=nxlnk; ixc=0; treeset !achsett ^achset]
  ([ixc<256] ([setno>0; inset !ixc !achset; "chartrans[CHR("]
  [number!ixc; ")]:= "; number!setno*2; ", "; ascii!0]))?
  [ixc@=ixc+1])* )?

```

```

["END ScannerFill;PROCEDURE Getoken;VAR ix,errno:INTEGER;ok:BOOLEAN;"]
["BEGIN ok:=TRUE;errno:=0;IF Debugging THEN writeC(''; ascii!34; ')END;"]
["REPEAT NextToken:=TokenPreview;TokenPreview:=0;"]
["IF LastCharacter>CHR(0)THEN writeC(LastCharacter)"]
["ELIF NOT Debugging THEN writeSln END;LastCharacter:=scannextch;"]
["IF EOF(source)THEN scannextch:=CHR(255) "]
["ELIF EOLN(source) THEN READLN(source);"]
["scannextch:=CHR(0)END ELSE READ(source,scannextch)END;"]
["ix:=CurrentState"; number!(hino*2+2)]
["+chartrans[scannextch];"]
["CASE StateTablePtr^[ix]OF"; ascii!0]
[newnewset !usacts ^usual; using=usual-4]
(using>usacts; number!using/4; " "; gots=mtset)
{::} [aclist=actsets; hitok=0; toktr=<>]
[aclist:<no aclnk actup >%listr; treeint !listr ^lino]
[newwinset !lino !usacts]
[actup:<no zlnk coder >%itmt; treeint !itmt ^item]
[[coder:<tr ..>%tkno; treeint !tkno ^tknn]
[[tknn>hitok; hitok=tknn; toktr=coder]]?
[notinset !item !gots] coder:flatten !env
[""]; addset !item !gots ^git; gots=git]]?
[actup=zlnk))* [aclist=aclnk))*
[hitok>0; ""]; toktr:flatten !env)?
["END;"; ascii!0; using=using-4))*
["0:END;CurrentState:=StateTablePtr^[ix+1];"]
["IF CurrentState=0 THEN TokenPreview:=-1 END;"; ascii!0]
["UNTIL NextToken<>0;IF Debugging THEN IF NextToken<0 "]
["THEN writeS(''; ascii!34; " ?? '')ELSE writeC('')"]
[ascii!34; '')END END END Getoken;"; ascii!0]
["PROCEDURE InitializeScanner;BEGIN "]
["ncharsets="; number!hino+1; ";TokenPreview:=0;"]
["scannextch:=CHR(0);LastCharacter:=CHR(0);"]
["CurrentState:=0;NewStateTable("]
[number!{doing+1}*(hino+1)*2; ");ScannerFill;"]
["NEW(StringTable);StringTable^[0]:=CHR(0);"]
["EndStrings:=1;Getoken END InitializeScanner;"; ascii!0]
["PROCEDURE MatchToken(tkno:INTEGER):BOOLEAN;"]
["BEGIN IF tkno=Nextoken THEN Getoken;"]
["RETURN TRUE ELSE RETURN FALSE END END MatchToken;"; ascii!0]
[matchtoks:<tk klnk ontree <no >%tknam >%entry] {/ders}
[entry:<tk ders >%tkno]
[treeint !tkno ^token; treeint !tknam ^toknam]

["PROCEDURE MatchTok; number!token]
[[ders#<>; "("] ders:paramlist !true !false ^osemi ["")"]
[":BOOLEAN;"]
["BEGIN IF NextToken="; number!token; " THEN "]
[[ders:<at dlnk vrv vty >%namt; ders#=dlnk]
["at"; treeint !namt ^name; spell!name; ":=sa"]
[number!token; spell!name; " "; ascii!0))*
["IF Debugging THEN writeS('="; spell!toknam; "');"]
ders:showdeprams ["writeSln END;"; ascii!0]
["Getoken;RETURN TRUE ELSE RETURN FALSE END END MatchTok; number!token; "];"]
[ascii!0; matchtoks=#klnk))*
[hicharset=hino; highstate=doing];

buildparser !indata:table ^outdata:table
-> [outdata=indata] indata:checkntcalls !inenv;

startoutput !name:int
-> ["MODULE "; spell !name]
[";FROM InOut IMPORT ReadChar,ReadLn,WriteStr,WriteLn;"];

finishoutput !goalname:int !name:int
-> ["BEGIN InitializeScanner; IF nt"; spell !goalname]
[" THEN WriteStr('Success')ELSE WriteStr('Failed')END END "]
[spell !name; "."];

```

```

globalname !name:int !itstype:tree
-> ["v"; spell !name; ":"] itstype:showtype [";"];

dolibrary !env:table
-> ;
    {output the predefined routines}

transformer

redecorate !decor:int !subtree:tree
-> <vr ...>
=> <vr subtree >%decor
-> <tk ...>
=> <tk subtree >%decor
-> <nt ...>
=> <nt subtree >%decor
-> <tv ...>
=> <tv subtree >%decor
-> <sa toknno xtyp>
=> <sa toknno subtree>%decor
-> <tr link fromst tost chrs acts >
    [intree !decor ^atree]
=> <tr link atree subtree chrs acts >;

retype !subtree:tree
-> <vr <>%decor
=> <vr subtree >%decor
-> <at next vrv <>%decor
=> <at next vrv subtree >%decor;

isdefined !defset:set
-> <at next vrv tipe >%idn
    next:isdefined !defset
    [/"undefined attribute "; spell !idn //; inset !idn !defset]
-> <no >;

checkntcalls !env:table
-> <tr link fst tst chs act >|<tk link fst chs >
    link:checkntcalls !env

-> <nc link <nt snd rcv trn >%name >
    link:checkntcalls !env
    [/"not a nonterminal: "; spell !name //]
    [from !name !env ^<nt <no inh der >%tgf rightp >]
    ([trn=<>; //spell !name; " is not a parser nonterminal"//; tgf=0]
    |[trn#<>; //spell !name; " is not a transformer nonterminal"//; tgf=1])
    snd:actualformal !inh !name
    rcv:actualformal !der !name

-> <nt link rpt >%name (do nonterminals as functions)
    [from !name !env ^<nt <no inh der >%tgf rightp >]
    makeforward !name !env !inh !der !tgf
    link:checkntcalls !env
    rpt:doflatten !name !env !inh !der !tgf
-> <> ;

makeforward !name:int !env:table !inh:tree !der:tree !tgf:int
-> ["PROCEDURE nt"; spell !name]
    ([inh:<>; der:<>; tgf=0]
    |[inh:<>; der:<>; tgf=1; "(thetree:tree)"]
    |[otherwise; "("]
    inh:paramlist !false !false ^isemi
    der:paramlist !true !isemi ^dsemi
    ([tgf=1; "(thetree:tree)"]|[tgf=0; ")"])))
    [":BOOLEAN;FORWARD;"];

actualformal !formal:tree !name:int

```

```

-> <>
    [/"too few actual attributes, calling "; spell !name //; formal:<>]
-> <at link exptree:<vr expty>%idn <>
    [formal:<at nextf vrv tipe >]
    [/"attribute type mismatch calling "; spell !name //]
    ( [expty=<>] exptree:rededcorate !idn !tipe
      | [otherwise; expty=tipe] )
    link:actualformal !nextf !name
=> <at link exptree tipe >
-> <at link <ca exptree:<tv expty>%idn mtt> <>
    [formal:<at nextf vrv tipe >]
    [/"attribute type mismatch calling "; spell !name //]
    ( [expty=<>] exptree:rededcorate !idn !tipe
      | [otherwise; expty=tipe] )
    link:actualformal !nextf !name
=> <at link <ca exptree mtt> tipe >
-> <at link exptree expty >
    [/"attribute type mismatch calling "; spell !name //]
    [formal:<at nextf vrv tipe >; expty#<>; expty=tipe]
    link:actualformal !nextf !name;

paramlist !needsvar:bool !insemi:bool ^outsemi:bool
-> <> [outsemi=insemi]
-> ident:<at link vrv tipe >%name
    link:paramlist !needsvar !insemi ^tsemi
    ([tsemi=true; ";"])?
    ([needsvar=true; "VAR "])?
    [outsemi=true]
    ident:shovar !false !false !true !true !empty ^xset;

showdeprams
-> ident:<at link vrv tipe >%name
    link:showdeprams
    ["Sho"] tipe:showtype ["("]
    ident:shovar !false !false !false !true !empty ^xset ["");"]
-> <> ;

doflatten !name:int !env:table !inh:tree !der:tree !tgf:int
-> <nt <no ...>%kind body >
    ["PROCEDURE nt"; spell !name; ";var ok:boolean;t0:tree;"]
    body:findvars !env !empty ^xset
    ["errno:INTEGER;PROCEDURE doerr;VAR ok:BOOLEAN;"]
    ["BEGIN ok:=TRUE;CASE errno OF 0:|"]
    body:finderrs !env !1 ^lasterr
    ["END;IF errno>0 THEN abortit ELSE errno:=0 END;"]
    ["END doerr; BEGIN ok:=TRUE;errno:=0;"]
    ( [kind=1; "t0:=thetree;"]|[otherwise; "t0:=NIL;"] )
    ["IF Debugging THEN writeS('+'; spell !name; ");"]
    ( [kind=1; "Shotree(t0);"] )?
    inh:showdeprams ["writeSln END;"]
    body:flatten !env
    ["IF NOT ok THEN doerr END;"]
    ["IF Debugging THEN writeS('-'; spell !name; ");"]
    ["IF ok THEN " der:showdeprams
    ["ELSE writeS(' ----')END;Shoboolean(ok);writeSln END;"]
    ["RETURN ok END nt"; spell !name; ";"];

findvars !env:table !indone:set ^outdone:set
-> <> | <ot > | <tr ...> [outdone=indone]
-> <al left right > | <ca left right > | <no left right >
    left:findvars !env !indone ^tdone
    right:findvars !env !tdone ^outdone
-> <st body > | <pl body > | <dl body tokn > | <er body >
    body:findvars !env !indone ^outdone
-> <tk params >
    params:argvars !env !true !indone ^outdone
-> <nt snd rcv tre> | <fn snd rcv >

```

```

    snd:argvars !env !false !indone ^tdone
    rcv:argvars !env !true !tdone ^outdone
-> <mt expt > | <xf expt >
    expt:expnvars !env !indone ^outdone
-> <as expt exty thev >
    thev:shovar !true !true !true !false !indone ^tdone
    expt:expnvars !env !tdone ^outdone ;

expnvars !env:table !indone:set ^outdone:set
-> <ca stm exp>
    stm:findvars !env !indone ^tdone
    exp:expnvars !env !tdone ^outdone
-> <bt expl exp2 >
    expl:bldvars !env !indone ^tdone
    exp2:expnvars !env !tdone ^outdone
-> (<or expl exp2 >|<an expl exp2 >|<eq expl exp2 >
    |<ne expl exp2 >|<ls expl exp2 >|<gr expl exp2 >|<ad expl exp2 >
    |<su expl exp2 >|<mp expl exp2 >|<dv expl exp2 >|<md expl exp2 >|
    expl:expnvars !env !indone ^tdone
    exp2:expnvars !env !tdone ^outdone
-> (<no exp >|<ng exp >|<nn exp >)
    exp:expnvars !env !indone ^outdone
-> <rt nod pth cmp <>>
    [outdone=indone]
-> <rt nod pth cmp var>
    var:shovar !true !true !true !false !indone ^outdone
-> (<|<cn ..>|<vr ..>|<at ..>|<gl ..>|<tv ..>|<sa ..>|
    [outdone=indone] ;

bldvars !env:table !indone:set ^outdone:set
-> <ca stm exp >
    stm:bldvars !env !indone ^tdone
    exp:expnvars !env !tdone ^outdone
-> exp:<bt ..>
    exp:expnvars !env !indone ^outdone
-> <> [outdone=indone] ;

argvars !env:table !uparo:bool !indone:set ^outdone:set
-> (exp:<vr ..>|exp:<at vlnx <> vty>|exp:<at vlnx <tk> vty>
    |exp:<gl ..>|exp:<sa ..>|
    [uparo=true]
    exp:shovar !true !true !true !false !indone ^outdone
-> <at nexta exptr expty >
    nexta:argvars !env !uparo !indone ^xdone
    ( [uparo=true]
    exptr:argvars !env !uparo !xdone ^outdone
    | [otherwise]
    exptr:expnvars !env !xdone ^outdone )
-> <ca exp stm >
    ( [uparo=true]
    exp:shovar !true !true !true !false !indone ^tdone
    stm:findvars !env !tdone ^outdone
    | [otherwise]
    exp:findvars !env !indone ^tdone
    stm:expnvars !env !tdone ^outdone )
-> <> [outdone=indone] ;

shovar !usesa:bool !needsem:bool !tytoo:bool !allvars:bool !indone:set ^outdone:set
-> <tv vty >%numb ["tv"; number !numb; outdone=indone]
    ( [tytoo=true; ":" ] vty:showtype
    ( [needsem=true; ";" ] ) ) ?
-> <gl vty >%name
    ( [allvars=true; "gl"; spell !name; addset !name !indone ^outdone]
    ([tytoo=true; ":" ] vty:showtype
    ( [needsem=true; ";" ] ) ) ?
    | [otherwise; outdone=indone] )
-> <at lnk vrv vty>%name

```

```

( [vrv:<tk>%toknno; toknno%-1; usesa=true;
  notinset !name*20+toknno !indone; "sa"; number !toknno;
  spell !name; addset !name*20+toknno !indone ^outdone]
  ([tytoo=true; ":" ] vty:showtype
    ([needsem=true; ";"])? )?
| [allvars=true]
  ["at" spell !name; addset !name !indone ^outdone]
  ([tytoo=true; ":" ] vty:showtype
    ([needsem=true; ";"])? )?
| [otherwise; outdone=indone]
-> <vr vty >%name
  ( [notinset !name !indone; addset !name !indone ^outdone]
    ["vr"; spell !name]
    ([tytoo=true; ":" ] vty:showtype
      ([needsem=true; ";"])? )?
    | [otherwise; outdone=indone]
-> <sa toknno vty>%name
  ( [notinset !name*20+toknno !indone;
    addset !name*20+toknno !indone ^outdone]
    ["sa"; number !toknno; spell !name]
    ([tytoo=true; ":" ] vty:showtype
      ([needsem=true; ";"])? )?
    | [otherwise; outdone=indone] );

showtype
-> <ty >%typeno
  ([typeno=1; "INTEGER"]
  |[typeno=2; "BOOLEAN"]
  |[typeno=3; "tree"]
  |[typeno=4; "tree"]
  |[typeno=5; "table"]
  |[typeno>5; "INTEGER"]);

finderrs !env:table !inerno:int ^outno:int
-> <> | <tr ..> | <ot > | <tk ..> | <nt ..> | <fn ..> | <mt ..> | <xf ..> | <as ..>
  [outno=inerno]
-> <al left right > | <ca left right > | <no left right >
  left:finderrs !env !inerno ^midno
  right:finderrs !env !midno ^outno
-> <st body > | <pl body > | <dl body tokn >
  body:finderrs !env !inerno ^outno
-> <er body >
  [number !inerno; ":" ]
  body:flatten !env
  ["|"; outno=inerno+1]
=> <er >%inerno ;

flatten !env:table
-> <> (no code)
-> <ca <> right >
  right:flatten !env
-> <ca left right >
  left:flatten !env
  ["IF ok THEN "]
  right:flatten !env
  ["ELSE doerr END;"]
-> <al left right >
  ["push(errno);errno:=0;push(Taken);Taken:=0;"]
  left:flatten !env
  ["IF NOT ok THEN doerr;ok:=TRUE;errno:=0;"]
  ["IF Taken>0 THEN SyntaxError END;"]
  ["IF Debugging THEN writeS(' ')END;"]
  right:flatten !env
  ["IF NOT ok THEN doerr;IF Taken>0 THEN SyntaxError END;"]
  ["END END;Taken:=Taken+pop();errno:=pop();"]
-> <st body >
  ["push(errno);push(Taken);WHILE ok DO "]

```

```

["errno:=0;Taken:=0;IF Debugging THEN writeS('')END;"]
body:flatten !env
["IF Debugging THEN writeS('')*')END;"]
["IF Taken>0 THEN IF ok THEN push(Taken+pop())"]
["ELSE SyntaxError END END END;"]
["Taken:=pop();errno:=pop();ok:=TRUE;"]
-> <pl body >
["push(errno);push(Taken);push(0);REPEAT "]
["Taken:=0;errno:=0;IF Debugging THEN writeS('')END;"]
body:flatten !env
["IF Debugging THEN writeS('')+')END;"]
["IF Taken>0 THEN IF ok THEN push(pop()+pop()+Taken);"]
["push(1) ELSE SyntaxError ELIF ok THEN "]
["push(pop()+1)END END UNTIL NOT ok;ok:=pop()>0;Taken:=pop();errno:=pop();"]
-> <dl body tokx>%tokn
["push(errno);push(Taken);push(0);REPEAT errno:=0;Taken:=0;"]
["IF Debugging THEN writeS('')END;"]
body:flatten !env
["IF Debugging THEN writeS('')$')END;push(Taken+pop)"]
["UNTIL NOT ok OR NOT matchtoken("; number !tokn; ");"]
["Taken:=pop;IF NOT ok THEN doerr;"]
["IF Taken>0 THEN SyntaxError END END;"]
["Taken:=Taken+pop;errno:=pop;"]
-> <tk <>>%tokn
["ok:=matchtoken("; number !tokn; ");"]
-> <tk params:<at ...>%tokn
["ok:=matchtok"; number !tokn; "("]
params:doargs !env !true !false !<> ^needs ^post
["");"]
post:flatten !env
-> <nt snd rcv tre>%name
["ok:=nt"; spell !name]
([snd:<>; rcv:<>; tre:<>; ";"]
|[otherwise; "("]
snd:doargs !env !false !false !<> ^sneed ^postx
rcv:doargs !env !true !sneed !postx ^rneed ^post
([tre#<>]
([rneed=true; ", "][rneed=false])
tre:shovar !true !false !false !true !empty ^xset)?
["");"])
post:flatten !env
-> <fn snd rcv >%name
["ok:=fn"; spell !name]
([snd:<>; rcv:<>; ";"]
|[otherwise; "("]
snd:doargs !env !false !false !<> ^sneed ^postx
rcv:doargs !env !true !sneed !postx ^rneed ^post
["");"])
post:flatten !env
-> <mt expt >
["ok:="]
expt:flattex !2 !env ^extx [";"]
-> <as expt <> thev >
thev:getvartype ^<ty>%exty
thev:shovar !true !false !false !true !empty ^xset [":="]
expt:flattex !exty !env ^extx [";"]
-> <as expt <ty>%exty thev >
thev:shovar !true !false !false !true !empty ^xset [":="]
expt:flattex !exty !env ^extx [";"]
-> <ot >%txtno
([txtno>0; "writeS('"; spell !txtno; "');"]
["IF Debugging THEN writeSln;writeS(''; spell !txtno; '')END;"]
|[txtno=0; "writeSln;"])
-> <xf body >
["replacetree(thatree,"]
body:flattex !3 !env ^extx [";"]
-> <tr ders >%toknot

```



```

[treeint !toknot ^tokno; "TokenPreview="; number!tokno; ";"]
-> <er ...>%errno
    ["if not ok then doerr;errno="; number !errno; ";"];

getvartype ^vartype:tree
-> <tv vartype>
-> <gl vartype>
-> <at lnk vrv vartype>
-> <vr vartype>
-> <sa tokno vartype>;

flattex !mustype:int !env:table ^istype:int
-> <rt root pth comp nvar> [from !l !env ^istype]
    ([nvar:<vr vty>][nvar:<at vlnx vlv vty>]
    [nvar:<gl vty>][nvar:<sa tokno vty>][otherwise:vty=<>])
    ([itype=vty; "treeparty("][otherwise; "treepart(")
    ([root=<>; "thetree"]
    [otherwise] root:shovar !true !false !false !true !empty ^xset)
    ["; treeint !pth ^path; number !path; "; treeint !comp ^con]
    [number !con; ";"]
    ([nvar=<>; "t0"]
    [otherwise] nvar:shovar !true !false !false !true !empty ^zset)
    [")"; istype=2; //"boolean type expected"//; mustype=2`mustype=0]
-> <bt <> <>>
    ["NIL"; istype=3; //"tree type expected"//; mustype=3`mustype=0]
-> <bt subs decor >ndn ["build("; number !ndn; ";"]
    [istype=3; //"tree type expected"//; mustype=3`mustype=0]
    decor:flattex !3 !env ^exty
    subs:buildtree !env ^nsubs [nnn=nsubs]
    ( [nsubs<8; ",NIL"; nsubs:=nsubs+1])* ["; number !nnn; ")"]
-> <nn expr:<vr exty> <> <ty>^toty>|<nn expr exty:<ty> <ty>^toty>
    exty:castype !toty ^itsty
    ["("; istype=toty; //"type mismatch"//; mustype=toty`mustype=0]
    expr:flattex !itsty !env ^extx [")"]
-> <or left right > ["("; //"boolean type expected"//; mustype=2`mustype=0]
    left:flattex !2 !env ^exty [")OR("]
    right:flattex !2 !env ^extz [")"; istype=2]
-> <an left right > ["("; //"boolean type expected"//; mustype=2`mustype=0]
    left:flattex !2 !env ^exty [")AND("]
    right:flattex !2 !env ^extz [")"; istype=2]
-> <no left > ["NOT("; //"boolean type expected"//; mustype=2`mustype=0]
    left:flattex !2 !env ^exty [")"; istype=2]
-> <eq left right > ["("; //"boolean type expected"//; mustype=2`mustype=0]
    left:flattex !0 !env ^exty ["="]
    right:flattex !0 !env ^exty [")"; istype=2]
-> <ne left right > ["("; //"boolean type expected"//; mustype=2`mustype=0]
    left:flattex !0 !env ^exty [")<("]
    right:flattex !0 !env ^exty [")"; istype=2]
-> <ls left right > ["("; //"boolean type expected"//; mustype=2`mustype=0]
    left:flattex !0 !env ^exty [")<("]
    right:flattex !0 !env ^exty
    [")"; istype=2; //"invalid magnitude compare not int"//; exty=1]
-> <gr left right > ["("; //"boolean type expected"//; mustype=2`mustype=0]
    left:flattex !0 !env ^exty [")>("]
    right:flattex !0 !env ^exty
    [")"; istype=2; //"invalid magnitude compare not int"//; exty=1]
-> <ad left right > ["("; //"integer type expected"//; mustype<2]
    left:flattex !1 !env ^exty [")+("]
    right:flattex !1 !env ^extz [")"; istype=1]
-> <su left right > ["("; //"integer type expected"//; mustype<2]
    left:flattex !1 !env ^exty [")-("]
    right:flattex !1 !env ^extz [")"; istype=1]
-> <mp left right > ["("; //"integer type expected"//; mustype<2]
    left:flattex !1 !env ^exty [")*("]
    right:flattex !1 !env ^extz [")"; istype=1]
-> <dv left right > ["("; //"integer type expected"//; mustype<2]
    left:flattex !1 !env ^exty [")DIV("]
    right:flattex !1 !env ^extz [")"; istype=1]

```

```

-> <md left right > ["("; //"integer type expected"; mustype<2]
  left:flattex !1 !env ^exty ["")MOD("]
  right:flattex !1 !env ^extz [")"; istype=1]
-> <ng left > ["-("; //"integer type expected"; mustype<2]
  left:flattex !1 !env ^exty [")"; istype=1]
-> thev:<vr <ty>%vty> | thev:<gl <ty>%vty> | thev:<at xx vrv <ty>%vty> |
  thev:<tv <ty>%vty> | thev:<sa toknno <ty>%vty>
  [/"identifier type mismatch"/; mustype=vty`mustype=0; istype=vty]
  thev:shovar !true !false !false !true !empty ^xset
-> <co <ty>%tipe >%idn
  [/"identifier type mismatch"/; mustype=tipe`mustype=0]
  [number !idn; istype=tipe]
-> <cn <ty>%2 >%0
  [/"boolean type expected"/; mustype=2`mustype=0][ "FALSE"; istype=2]
-> <cn <ty>%2 >%1
  [/"boolean type expected"/; mustype=2`mustype=0][ "TRUE"; istype=2]
-> <cn <ty>%4 >
  [/"set type expected"/; mustype=4`mustype=0][ "NIL"; istype=4]
-> <cn <ty>%5 >
  [/"symbol table type expected"/; mustype=5`mustype=0][ "NIL"; istype=5]
-> <cn <ty>%1 >%val
  [/"integer type expected"/; mustype<2][number !val; istype=1]
-> <cn <ty>%cty >%val
  [/"invalid constant type"/; mustype=cty`mustype=0; cty>5]
  [number !val; istype=cty] ;

doargs !env:table !uparo:bool !insemi:bool !repost:tree
  ^outsemi:bool ^post:tree
-> <> [outsemi=insemi; post=repost]
-> <at nexta exptr expty:<ty >%tyno >
  nexta:doargs !env !uparo !insemi !repost ^xsemi ^postx [outsemi=true]
  ( [xsemi=true; ", "])?
  ( [uparo=true] exptr:splitarg !postx ^post
  | [otherwise; post=postx] exptr:flattex !tyno !env ^xty );

splitarg !pre:tree ^post:tree
-> <ca exptr thep >
  exptr:shovar !true !false !false !true !empty ^xset [post=<ca pre thep >]
-> thev:<vr ...> | thev:<gl ...> | thev:<at ...> | thev:<tv ...> | thev:<sa ...>
  thev:shovar !true !false !false !true !empty ^xset [post=pre];

buildtree !env:table ^nsubs:int
-> <bt <> <>> [nsubs=0]
-> <ca next extr >
  next:buildtree !env ^nmos ["", "]; nsubs=nmos+1]
  extr:flattex !3 !env ^xty ;

castype !wanty:int ^typeno:int
-> <ty >%typeno
  ([typeno=1; "int"]|[typeno=2; "bool"]|[typeno=3; "tree"]
  |[typeno=4; "tree"]|[typeno=5; "tree"]|[typeno>5; "int"])
  ([wanty=1; "int"]|[wanty=2; "bool"]|[wanty=3; "tree"]
  |[wanty=4; "tree"]|[wanty=5; "tree"]|[wanty>5; "int"]);

deleteitem (delete from list, replace with its link)
-> <no link aset >
=> link
-> <tr link fromst tost chrs acts >
=> link;

catlist !tail:tree ^result:tree (cats tree to tail)
-> <> [result=tail]
-> <no link code >%deco
  link:catlist !tail ^midlist [result=<no midlist code >%deco];

end TagGrammar.

```

附录 C Itty Bitty 栈机器的指令集

Itty Bitty 栈机器是一种虚构的栈计算机^①，它有一个简单的指令集，并且仅有一种寻址模式。这使得编译 Modula-2 这类高级语言时更容易为 IBSM 生成目标代码，同时也使得一些代码优化问题更清晰。从实际效果看，该计算机有三个寄存器，这些寄存器都不是直接可编程的（即没有专门的装入或存储指令用于改变寄存器的内容）。这三个寄存器分别是程序计数器、栈指针以及帧指针。第四个寄存器（Limit）用于保护用户免受栈溢出的危害，但它通常是不可改变的。在启动时，所有四个寄存器均从内存设置了一个初始值。

当程序执行时，程序计数器（Program Counter，简称 PC）沿机器代码前进。有三条指令可以用其他方式修改程序计数器的内容，从而改变指令的执行序列。

栈指针（Stack Pointer，简称 SP）总是指向表达式和控制栈的顶部。不同于许多现代计算机，IBSM 的栈朝着递增的内存地址沿正方向增长。大多数指令都会导致 SP 随栈的增长或缩小而分别递增或递减。有两条指令作为进入和退出一个过程的一部分，它们对 SP 的影响更关键。

帧指针（Frame Pointer，简称 FP）是两条内存引用指令的基地址。作为进入和退出一个过程的一部分，FP 也会随之隐式地被改变；除此之外，没有其他方法可改变 FP。

IBSM 的设计目标是其操作方式与任意字长的计算机尽量接近，惟一例外是对于更长的机器字，可将多条指令压缩在单个字中。每一指令占 5 个位，因而可将 3 条指令压缩在一个 16 位的字中（第 16 位还可容纳仅由一个有效位组成的第 4 条指令），如图 C-1 所示。如果字长为 32 位，则编译程序可将 6 条或 7 条指令压缩到一个字中。在每一个字仅含一条指令的情况下，IBSM 也可运行良好，如第 6 章所述。当每一个字压缩了多条指令时，先执行该字中最低有效的 5 位，然后执行下一个 5 位，如此类推，直至不再有非 0 的指令。分支语句与过程调用（以及返回）总是导致程序从寻址找到的字的第一条指令开始继续执行。IBSM 没有类似的字节寻址模式；字是原子的。

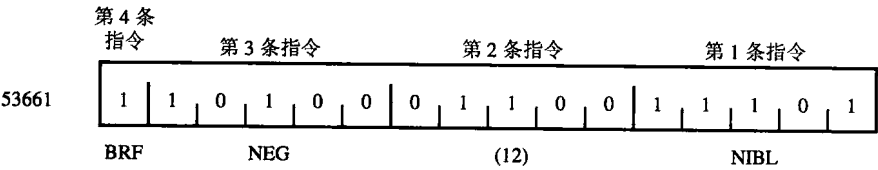


图 C-1 IBSM 指令字（16 位版本）

程序启动时，IBSM 从内存的（绝对）位置 0 读入一个地址，并将它作为栈指针的起始值。然后 IBSM 从这一初始栈的顶部弹出三个字（倒计数，因而这三个字的位置是 n 、 $n-1$ 和 $n-2$ ），并将它们分别设置为栈的界限（以避免栈溢出）、帧指针的初始值以及程序计数器的初始值。在代码清单 6.1 中，它们被设置为：

SP	3	（初始时，在弹出之前）
Limit	100	
FP	0	

① 作者 T. Pittman 关于 IBSM 的最新描述可参见：<http://www.ittybittycomputers.com/Courses/Prior/CC/IBSM.htm>。——译者注

PC 100
SP 0 (弹出其余 3 个字之后)

如果需要运行更大的程序 (例如需要比 100 个字更多的栈增长), 则可修改目标文件中的第一行, 用一个更大的数字系统地替换这些为 100 的值。

一些指令代码被保留以备多任务的应用, 并且它们与编译程序设计原理没有关系。另一些指令代码被保留是为了未来的研究。表 C-1 定义了剩下的 28 个指令, 表中的指令代码以十进制表示。

表 C-1 IBSM 指令集

代 码	助记符	描 述
00	Nop	无操作 (No operation)。在一个部分填充的指令字中用作填充符
01	BrFalse	为假时分支 (Branch if False)。从栈中弹出两个数字, 如果第二个数字为 0, 则将第一个数字累加到程序计数器 PC, 并继续执行这个字的第一条指令; 否则执行指令序列中的下一指令。这可以是压缩在 16 位指令字中的第四条指令
02		(保留)
03	Call	调用 (Call) 一个 (绝对) 地址位于栈顶的过程, 其调用方式是交换栈顶与 PC 的内容。一个未使用 ENTER 的过程也可用这一指令返回, 可是如此一来则必须由调用者负责丢弃栈顶元素。作为返回地址压入栈中的 PC 值总是下一指令字的地址, 即便在当前字中仍有未用的指令
04	Enter	进入 (Enter) 一个过程。弹出栈顶字 n , 代之以帧指针 (动态链, 指向一个在其头部使用了它的过程) 的当前内容, 让帧指针重新指向静态链 (假设位于动态链之下的第二个字), 然后调整栈指针向上移动 n 个字 (以保留局部变量的空间)
05	Exit	退出 (Exit) 一个由 Enter 指令进入的过程。从栈顶弹出值 n , 设置 SP 指向动态链 (即 FP 中的值再加 2), 然后弹出动态链存入 FP 中, 弹出返回地址存入 PC 中, 最后再弹出并丢弃 n 个字 (即过程的参数, 此时不再需要)
06		(保留)
07		
08	Dupe	复制 (Duplicate) 栈顶元素, 其方法为将栈顶字的一个副本压入栈中
09	Swap	交换 (Swap) 栈顶的两个字, 颠倒它们的相对位置
10		(保留)
11	Mpy	将栈顶的两个字相乘 (Multiply), 用单个字的积取代它们。如果乘法结果大于一个字, 则超出的高位将丢失
12	Add	将栈顶的两个字相加 (Add), 用它们的和取代它们。如果加法结果溢出, 则将有错误的符号
13	Xor	将栈顶的两个字执行逐位的异或 (exclusive-or), 用结果取代它们
14	Or	将栈顶的两个字执行逐位的逻辑或 (inclusive-Or), 用结果取代它们
15	And	将栈顶的两个字执行逐位的逻辑与 (logical And), 用结果取代它们
16	EQUAL	比较栈顶的两个字是否相等 (EQUAL); 如果相等则用真 (1) 取代它们, 否则用假 (0) 取代它们
17	Less	比较栈顶的两个字是否有小于 (Less) 关系; 如果第一个字大于第二个字则用真取代它们, 否则用假取代它们
18	Greater	比较栈顶的两个字是否有大于 (Greater) 关系; 如果第一个字小于第二个字则用真取代它们, 否则用假取代它们
19	Not	将栈顶字的每一位求反
20	Negate	用栈顶字的二进制补码负数 (Negative) 取代栈顶字
21		(保留)
22		
23		
24	Stop	停止 (Stop) IBSM 的执行。这条指令通常留作出错时停机之用

(续)

代 码	助记符	描 述
25	Global	从栈顶字中减去 FP。如果栈顶字是内存中的一个绝对地址，这会将它转换为一个相对于 FP 的地址，从而装入和存储指令可正确地工作
26	Store	存储 (Store) 一个变量。弹出栈顶的两个值，并将第一个值存储到由第二个值与 FP 之和寻址的内存位置。该指令为相对于 FP 寻址的局部变量提供了一种简单的访问方式
27	Load	装入 (Load) 一个变量。弹出栈顶值，并用由该值与 FP 之和寻址的内存位置中的内容取代它。
28	LoadConstant	装入一个常量 (Load Constant)。将 PC 所指向的字的值压入栈中，并让 PC 递增。注意当前指令字中另外的指令将继续执行，但下一指令字将跟随在常量字之后
29	Nibble	装入短常量。将当前指令字之后 5 个位的值作为一个字压入栈中，并且不将该部分作为一条指令执行。如果在 5 个位的常量之后有另外的指令，则继续执行这些指令，否则继续执行 PC 指向的下一个字
30	Zero	装入 0 (Zero)。将 0 (假) 作为一个字压入栈中
31	One	装入 1 (One)。将 1 (真) 作为一个字压入栈中

IBSM 已有一个解释程序，采用 ISO Pascal 语言的一个适度可移植的子集编写。采用一些不太正统的技巧可提高几个过程的执行速度，这些过程已作了标记。该解释程序接受一个包含 IBSM 代码的文本文件，并装入十进制数字表示的信息。除非显式地改变了装入地址，否则从地址 0 开始装入虚拟机的内存，并连续地顺序装入，每一个字表示一个数字。

数字“-1”作为一个转义代码使用。如果一个值后面跟着另一个“-1”，则该值存入下一个字并继续装入过程。如果下一个值是一个正数且是一个有效地址，则从文件中该值表示的地址开始继续装入；如果下一个值大于最大的合法地址，这表明文件已结束，则通过从地址 0 取出 SP 并弹出剩余的寄存器开始执行程序。

如果跟在转义代码“-1”后面的是一个负数，但不是“-1”，则使用该数字低 5 位中的 4 位设置一些支持追踪的标志，然后正常地开始执行。有效的追踪支持位包括以下标志：

- 0 (1) (保留)
- 1 (2) 追踪对内存的写操作 (ST 指令)
- 2 (4) 追踪序列执行中的任何修改
- 3 (8) 仅追踪过程的进入与退出
- 4 (16) 追踪每一指令

当该负数的某一位为 0 时（即用到了补码），相应的追踪支持位被设置；因而值-31 开启了所有的追踪，而-9 则仅追踪过程的进入与退出。为设置某一特定的追踪级别，应加上选中的位的值，并从求和结果中减去-1。

指令追踪结果显示了一条指令的“周期”计数和当前指令的地址，接着是 SP 和栈顶的值，以及指令执行后的 FP。顺序改变和内存写操作的追踪将找出所涉及的各个地址；对于追踪写操作的情况，还包括存储到那里的数据。

在一个过程入口的开始处可生成代码清单 6.3 所包含的十进制代码，用于构建一张 Display 表。代码清单 C.1 展示了该操作的机器码助记符。

代码清单 C.1 为 IBSM 栈构建一个 Display 表的代码

28828	Entry:	LoadCon	#nvars	; 局部变量的数目
		Enter		; 在栈中分配变量的空间

		LoadCon	#lex	; Display 表中单元的数目
44968	Loop:	Dupe		
		Nibble	#11	; 至 Done 的偏移量
		BrFalse		; 若无新的单元要构建则退出
13288		Dupe		
		One		; 剩余副本的计数器递增,
		Add		; 从而最后一个项目立即退出
30		Zero		
21481	UpLevel:	Swap		; 为该项目寻找静态链
		One		; 计数器递减, 为 0 则跳转到 GotIt
		Negate		
268		Add		
		Dupe		
1149	Here:	Nibble	#3	
		BrFalse		
26473		Swap		; 装入下一静态链 (父帧)
		Load		
		Global		
7102		Zero		; 无条件分支跳回 UpLevel
		Nibble	#6	
52		Negate		
		BrFalse		
32044	GotIt:	Add		; 通过累加到栈顶来处置 0
		Swap		; 在计数器之下插入该链
		One		; 计数器递减, 然后跳转到 Loop
31124		Negate		
		Add		
		Zero		
53661		Nibble	#12	
		Negate		
		BrFalse		
	Done:			

附录 D 四种计算机的代码生成表

第 9 章包括了一个表驱动模块的样板源代码，该模块将 Itty Bitty 栈机器代码翻译为四种流行的计算机的寄存器代码。本附录用两张大表列出了这些数据。这些数据表应按垂直方向阅读，即表中的每一列表示一种计算机对应的数据（忽略另外三列数据）。这些数据表中未包含过程的入口和出口块，也未包含调用一个过程的代码（留待读者作为练习）。

表 D-1 四种流行计算机的 TargetMachine 代码索引

OpCodeIndex:	(11)	(86)	(370)	(68000)
(Nop)	0,0,0,	0,0,0,	0,0,0,	0,0,0,
	0,0,0,	0,0,0,	0,0,0,	0,0,0,
(Load)	0,7,15,	0,7,14,	0,7,14,	0,8,16,
	23,31,39,	22,30,0,	23,32,0,	24,32,39,
(Store)	0,0,57,	0,0,38,	0,0,39,	0,0,47,
	23,0,0,	22,0,0,	23,0,0,	24,0,0,
(Cmpr)	0,65,73,	0,46,54,	0,0,48,	0,55,63,
	0,81,0,	0,62,0,	0,57,0,	0,71,0,
(Add)	0,89,97,	0,70,78,	0,0,64,	0,79,87,
	0,105,0,	0,86,0,	0,73,0,	0,95,0,
(Subt)	0,113,121,	0,94,102,	0,0,80,	0,103,111,
	0,129,0,	0,110,0,	0,89,0,	0,119,0,
(Mlpy)	0,137,145,	0,0,118,	0,0,96,	0,127,135,
	0,153,0,	0,126,0,	0,105,0,	0,143,0,
(Neg)	0,0,0,	0,0,0,	0,0,0,	0,151,159,
	0,161,0,	0,134,0,	0,112,0,	0,167,0,
(And)	0,0,0,	0,142,150,	0,0,119,	0,175,183,
	0,0,0,	0,158,0,	0,128,0,	0,191,0,
(Or)	0,169,177,	0,166,174,	0,0,135,	0,199,207,
	0,185,0,	0,182,0,	0,144,0,	0,215,0,
(Jump)	0,0,193,	0,0,190,	0,0,151,	0,0,223,
	0,0,0,	0,0,0,	0,0,0,	0,0,0,
(Bcc)	0,0,200,	0,0,197,	0,0,159,	0,0,230,
	0,0,0,	0,0,0,	0,0,0,	0,0,0,

表 D-2 四种流行计算机的 TargetMachine 代码

OpCode Table values:	(11)	(86)	(370)	(68000)
Nop(all modes)	2,0,0,	2,0,0,	2,0,0,	
	0,0,1,	0,0,1,	0,0,1,	0,0,2,
	160,	144,	7,	78,113,
Load r,#con	4,0,1,	3,0,1,	4,1,16,	6,0,2,
	2,2,2,	1,2,1,	2,2,1,	2,4,2,
	192,21,	184,	65,	32,60,
Load r,mem	4,0,1,	4,1,8,	4,1,16,	4,0,2,
	2,2,2,	2,2,2,	2,2,3,	2,2,2,
	64,23,	139,134,	88,0,208,	32,46,

(续)

OpCode Table values:	(11)	(86)	(370)	(68000)
Load r,@r	2,0,1,	2,1,8,	4,1,16,	4,0,2,
	0,64,2,	1,1,2,	1,1,3,	2,16,2,
	0,18,	139,4,	88,0,208,	32,54,
Load r,r	2,0,1,	2,1,8,	2,1,16,	2,0,2,
	0,64,2,	1,1,2,	1,1,1,	1,1,1,
	0,16,	139,192,	24,	32,
Load r,cc	12,11,1,			2,1,1,
	0,128,12,			0,1,2,
	2,0,38,			80,192,
	10,142,			
	10,1,1,			
	38,10,			
	128,21,			
Store r,mem	4,0,64,	4,1,8,	4,1,16,	4,1,1,
	2,2,2,	2,2,2,	2,2,3,	2,2,2,
	53,16,	137,134,	80,0,208,	45,64,
Cmpr r,#con	4,0,1,	4,1,1,		6,0,2,
	2,2,2,	2,2,2,		2,4,2,
	192,37,	129,248,		176,188,
Cmpr r,mem	4,0,1,	4,1,8,	4,1,16,	4,0,2,
	2,2,2,	2,2,2,	2,2,3,	2,2,2,
	64,45,	59,134,	89,0,208,	176,174,
Cmp	2,0,1,	2,1,8,	2,1,16,	2,0,2,
	0,64,2,	1,1,2,	1,1,1,	1,1,2,
	0,32,	59,192,	25,	176,128,
Add r,#con	4,0,1,	4,1,1,		6,0,2,
	2,2,2,	2,2,2,		2,4,2,
	192,101,	129,192,		208,188,
Add r,mem	4,0,1,	4,1,8,	4,1,16,	4,0,2,
	2,2,2,	2,2,2,	2,2,3,	2,2,2,
	64,109,	3,134,	90,0,208,	208,174,
Add r,r	2,0,1,	2,1,8,	2,1,16,	2,0,2,
	0,64,2,	1,1,2,	1,1,1,	1,1,2,
	0,96,	3,192,	26,	208,128,
Subt r,#con	4,0,1,	4,1,1,		6,0,2,
	2,2,2,	2,2,2,		2,4,2,
	192,229,	129,232,		144,188,
Subt r,mem	4,0,1,	4,1,8,	4,1,16,	4,0,2,
	2,2,2,	2,2,2,	2,2,3,	2,2,2,
	64,237,	43,134,	91,0,208,	144,174,
Subt r,r	2,0,1,	2,1,8,	2,1,16,	2,0,2,
	0,64,2,	1,1,2,	1,1,1,	1,1,2,
	0,224,	43,192,	27,	144,128,
Mlpy r,#con	4,0,1,			6,0,2,
	2,2,2,			2,4,2,
	192,117,			193,252,

(续)

OpCode Table values:	(11)	(86)	(370)	(68000)
Mlpy r,mem	4,0,1,	4,1,8,	4,1,16,	4,0,2,
	2,2,2,	2,2,2,	2,2,3,	2,2,2,
	64,125,	247,166,	92,0,208,	193,238,
Mlpy r,r	2,0,1,	2,1,8,	2,1,16,	2,0,2,
	0,64,2,	1,1,2,	1,1,1,	1,1,2,
	0,112,	247,224,	28,	193,192,
Neg r	2,0,1,	2,1,8,	2,1,16,	2,0,2,
	0,0,2,	1,1,2,	1,1,1,	1,1,2,
	0,11,	43,192,	, 27,	144,128,
And r,#con		4,1,1,		6,0,2,
		2,2,2,		2,4,2,
		129,224,		192,188,
And r,mem		4,1,8,	4,1,16,	4,0,2,
		2,2,2,	2,2,3,	2,2,2,
		35,134,	84,0,208,	192,174,
And r,r		2,1,8,	2,1,16,	2,0,2,
		1,1,2,	1,1,1,	1,1,2,
		35,192,	20,	192,128,
Or r,#con	4,0,1,	4,1,1,		6,0,2,
	2,2,2,	2,2,2,		2,4,2,
	192,85,	129,200,		128,188,
Or r,mem	4,0,1,	4,1,8,	4,1,16,	4,0,2,
	2,2,2,	2,2,2,	2,2,3,	2,2,2,
	64,93,	11,134,	86,0,208,	128,174,
Or r,r	2,0,1,	2,1,8,	2,1,16,	2,0,2,
	0,64,2,	1,1,2,	1,1,1,	1,1,2,
	0,80,	11,192,	22,	128,128,
Jum mem	4,0,0,	3,0,0,	4,0,0,	4,0,0,
	2,2,1,	1,2,1,	2,2,2,	2,2,1,
	119,	233,	71,240,	96,
Bcc c,mem	6,0,128,	3,0,1,	4,1,16,	4,0,1,
	2,2,3,	1,2,1,	2,2,1,	2,2,1,
	2,0,119;	112;	71;	96;